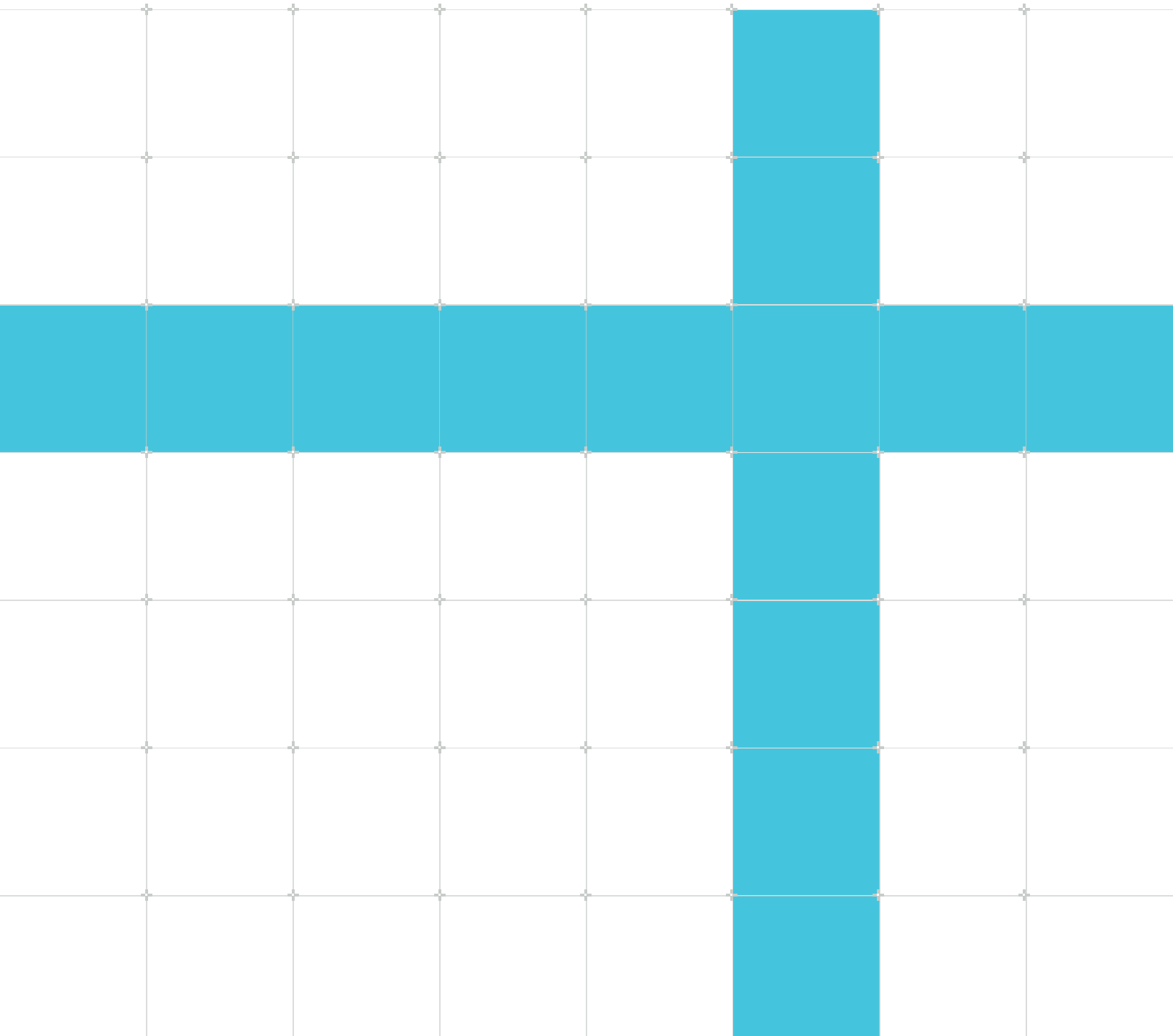
arm

SME Programmer's Guide

Version 1.0

Issue 01
109246_0100_01_en



SME Programmer's Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	23 May 2024	Non-Confidential	First release

Proprietary Notice

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

As the technology is evolving faster than ever before in the digital world, so is the demand for accelerated computing. The Arm architecture is evolving continuously to cope with the ever-rising demand for computing complex data, making it the perfect choice for emerging digital technology. This guide describes the Scalable Matrix Extensions, SME and SME2. This extension is introduced into Armv9-A to improve the efficiency and performance of matrix operations.

The Armv7-A Advanced SIMD Extension enabled parallel data-processing on multiple lanes within the fixed length 64-bit and 128-bit vector registers. With SIMD, the explicit data parallelization technique helped improve performance in applications involving large amounts of data processing. Industry applications involving large amounts of data processing demanded a larger vector length. Arm adopted a novel approach and created a unique instruction set that scaled to different vector lengths called the Scalable Vector Extension (SVE). SVE is an architectural extension to Armv8-A. SVE enabled implementation defined vector register length which must be a power of two, from 128 bits to 2048 bits.

The vector length agnostic (VLA) coding style of the SVE architecture enables binaries, built for SVE, to be portable across systems with different vector lengths. The programs built for the SVE architecture can be executed on any Arm-based systems with SVE and are able to use the full vector length across different platforms with different vector lengths without the need to re-compile the source code. The SVE architecture significantly enhanced the computing performance for 'High Performance Computing', making Arm the perfect choice for HPC designs.

The SVE2 architecture (a superset of SVE), introduced as part of Armv9-A, further extended the SVE ISA to support DSP/Media processing which expands its scope from supercomputers to personal devices.

The vectorization techniques introduced in SVE and SVE2 improved the performance of complex data computation workloads significantly. However, efficient matrix computation commonly used in Artificial Intelligence (AI) and Computer Vision remained a challenge. The architectural challenge for matrices computation is to enable efficient data parallelism with minimal memory bandwidth to achieve the best balance of compute ability and memory bandwidth. The Scalable Matrix Extension (SME) introduced in this guide is an architectural solution by Arm to accelerate matrix operations.

1.1 The Scalable Matrix Extensions

The Scalable Matrix Extensions (SME and SME2) define:

- Streaming SVE mode
- ZA storage

1.1.1 Streaming SVE mode and ZA storage

The features of Streaming SVE mode and ZA storage are as follows:

- When the Processing Element (PE) is in Streaming SVE mode, the Streaming SVE register state is defined. The Streaming SVE register state consists of:
  - Streaming vector registers Z0-Z31
  - Streaming predicate registers P0-P15
- Streaming SVE mode supports execution of a subset of SVE2 instructions with SME defined vector length known as Streaming SVE Vector Length (SVL). When the PE is in Streaming SVE mode, the effective SVE Vector Length (VL) is equal to a power of two in the range 128-2048 bits inclusive.

  In a vector of SVL bits:
  - SVL_B is the number of 8-bit elements
  - SVL_H is the number of 16-bit elements
  - SVL_S is the number of 32-bit elements
  - SVL_D is the number of 64-bit elements
  - SVL_Q is the number of 128-bit elements
- The ZA storage is an architectural register state consisting of a two-dimensional ZA array of [SVL_B x SVL_B] bytes
- The ZA array can be accessed as vectors of SVL bits represented by ZA[N], where N is in the range 0 to SVL_B-1 inclusive
- An elementwise vector access to the ZA array is shown in this document by appending an element size qualifier and a vector index "[N]" to the ZA array name where N is in the range 0 to SVL_B-1 inclusive, as follows:
  - ZA.B[N] represents an 8-bit element vector access to the ZA array
  - ZA.H[N] represents an 16-bit element vector access to the ZA array
  - ZA.S[N] represents an 32-bit element vector access to the ZA array
  - ZA.D[N] represents an 64-bit element vector access to the ZA array
  - ZA.Q[N] represents an 128-bit element vector access to the ZA array

  A sub-array of elements within the ZA array can be accessed as a tile. A ZA tile is a square, two-dimensional sub-array of elements within the ZA array. As the architecture defines ZA tile as a square, the ZA array is treated as containing one or more ZA tiles depending on the element size with which the ZA array is accessed. A ZA tile is represented by appending the tile number to the ZA name followed by an element qualifier.

  Also, a one-dimensional set of horizontally or vertically contiguous elements within a ZA tile can be accessed as a ZA tile slice. An access to horizontal tile slices is indicated by an "H" suffix on the ZA tile name. An access to vertical tile slices is indicated by a "V" suffix on the ZA tile name.

Accessing an 8-bit element ZA tile

There is a single tile ZA0.B which consists of [$SVL_B \times SVL_B$] 8-bit elements and occupies all of the ZA storage.

Accessing a 16-bit element ZA tile

There are two tiles, ZA0.H and ZA1.H. Each tile consists of [$SVL_H \times SVL_H$] 16-bit elements and occupies half of the ZA storage.

Accessing a 32-bit element ZA tile

There are four tiles, ZA0.S, ZA1.S, ZA2.S, and ZA3.S . Each tile consists of [$SVL_S \times SVL_S$] 32-bit elements and occupies a quarter of the ZA storage.

Accessing a 64-bit element ZA tile

There are eight tiles, ZA0.D, ZA1.D, ZA2.D, ZA3.D, ZA4.D, ZA5.D, ZA6.D, and ZA7.D. Each tile consists of [$SVL_D \times SVL_D$] 64-bit elements and occupies an eighth of ZA storage.

Accessing a 128-bit element ZA tile

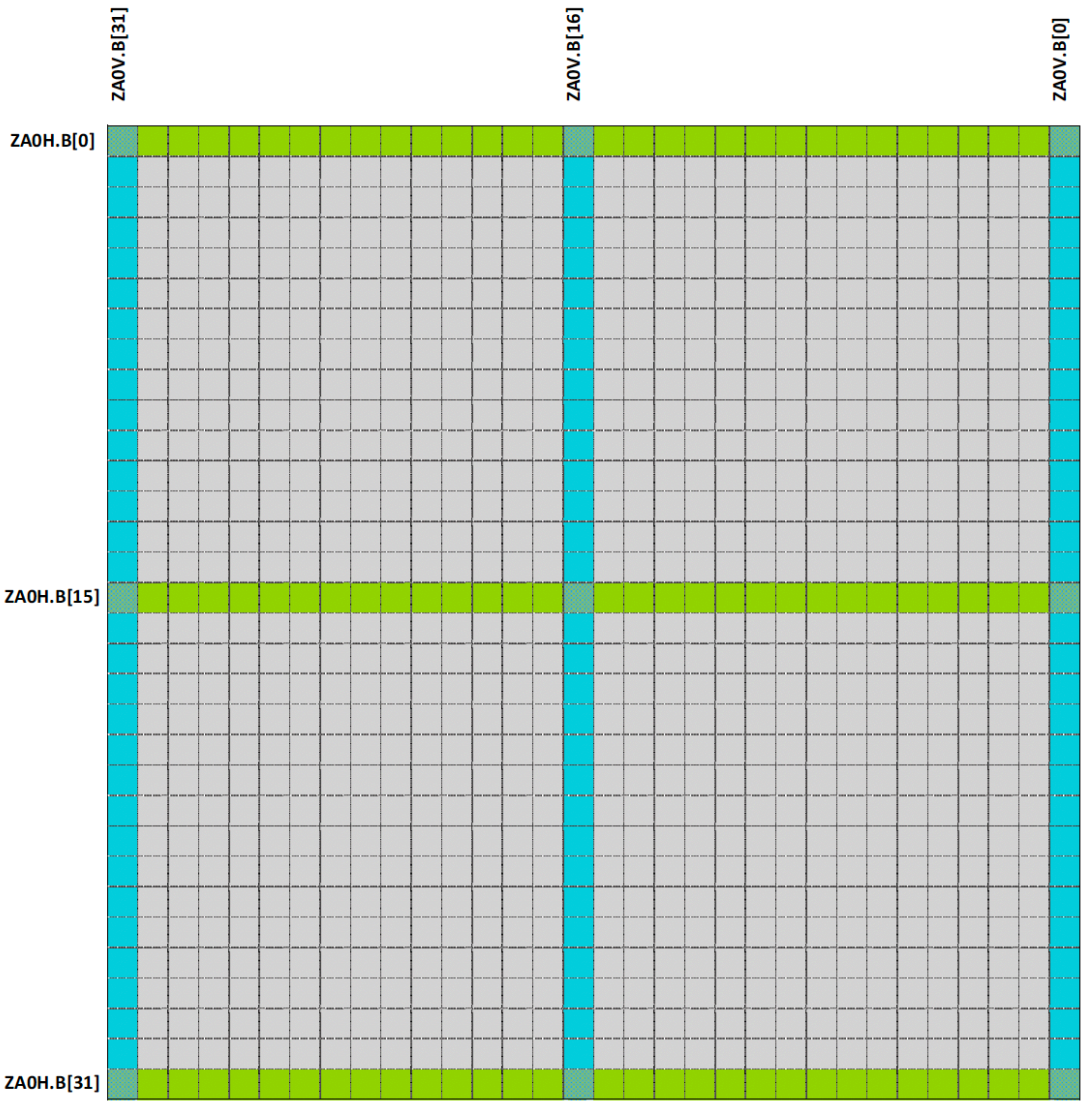
There are sixteen tiles, ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, and ZA15.Q. Each tile consists of [$SVL_Q \times SVL_Q$] 128-bit elements and occupies a sixteenth of ZA storage.

Figure 1-1 shows tile ZA0.B when SVL is 256 bits consisting of SVL_B x SVL_B 8-bit elements.

In Figure 1-1:

- ZA0H.B[0], ZA0H.B[15], ZA0H.B[31] indicates horizontal tile slice selection of 8-bit element ZA tile
- ZA0V.B[0], ZA0V.B[15], ZA0V.B[31] indicates vertical 8-bit element ZA tile slice selection

Figure 1-1: 8-bit element ZA tile, SVL = 256 bits



2. SME Overview

This chapter gives an overview of SME and SME2.

2.1 SME and SME2

The Scalable Matrix Extension (SME) is an extension to Armv9-A architecture. It adds a new architecture state: ZA storage. It introduces a new execution mode: Streaming SVE mode in which the new SME instructions and a subset of SVE2 instructions can be executed. The SME instructions operating on ZA include:

- Outer product computation with results in ZA tile
- Insert/extract ZA tile slice from/to Z vector register
- Load and store of ZA tile slice

The above SME instructions enable matrix operations, such as multiplication, inversion, and on-the-fly transposition. These instructions are useful in digital filtering, linear equation solvers, and convolutions. SME further extends a PE's Process state or PSTATE with the SM and ZA fields. For more details refer sections Streaming SVE mode and SME ZA storage.

The Scalable Matrix Extension version 2 (SME2) extends the SME architecture by accelerating vector operations to increase the number of applications that can benefit from the computational efficiency of SME, beyond its initial focus on outer products and matrix-matrix multiplication.

SME2 extends SME by introducing multi-vector data-processing instructions, load to and store from multi-vectors, and a multi-vector predication mechanism.

Additional architectural features of SME2 include:

- Multi-vector multiply-accumulate instructions, with Z vectors as multiplier and multiplicand inputs and accumulating results into ZA array vectors, including widening multiplies that accumulate into more vectors than they read
- Multi-vector load, store, move, permute, and convert instructions, that use multiple SVE Z vectors as source and destination registers to pre-process inputs and post-process outputs of the ZA-targeting SME2 instructions
- "Predicate-as-counter", an alternative predication mechanism is added to the original SVE predication mechanism, to control operations performed on multiple vector registers
- Compressed neural network capability using dedicated lookup table instructions and outer product instructions that support binary neural networks

SME2 adds a 512-bit architectural register ZT0, that supports the lookup table feature.

2.1.1 If SME and SME2 are supported

Implementation of SME is represented by the feature FEAT_SME which is an optional extension from Armv9.2-A. The ID_AA64PFR1_EL1.SME system register field contains the value 1 or higher to indicate the presence of FEAT_SME.

Implementation of SME2 is represented by the feature FEAT_SME2 which is an optional extension from Armv9.2-A. The following fields show the presence of FEAT_SME2:

- ID_AA64PFR1_EL1.SME contains the value 2 or higher, and
- ID_AA64SMFR0_EL1.SMEver contains the 1 or higher

FEAT_SME2 requires FEAT_SME.

System registers (ID_AA64PFR1_EL1 and ID_AA64SMFR0_EL1) might not be directly accessible by an application program. The operating system might provide this information to an application via an alternative means, such as the Linux hwcap mechanism.

2.1.2 SME2 lookup table

The 512-bit architectural register ZT0, supports the lookup table feature. The ZT0 contains a 16-entry lookup table. Each entry is 32 bits wide and can hold an element that can be 8 bits, 16 bits, or 32 bits. The ZT0 register can be loaded from memory and stored to memory.

Loading the ZT0 register

LDR (table), Load ZT0 register from the memory address provided in the 64-bit scalar base register. For example: - `LDR ZT0, [X0].`

Storing the ZT0 register

STR (table), Store ZT0 register to the memory address provided in the 64-bit scalar base register. For example: - `STR ZT0, [X0].`

Lookup in the ZT0 register

LUTI2 and LUTI4, Lookup table read instructions, copy 8-bit, 16-bit or 32-bit elements from ZT0 to one or more of the destination vector registers using packed indices from a segment of the source vector register. The indices can be 2 bits (LUTI2) or 4-bits (LUTI4). When using 2-bit indices, only the first four entries of the ZT0 table are accessible. When using 4-bit indices, all sixteen entries of the ZT0 table are accessible.

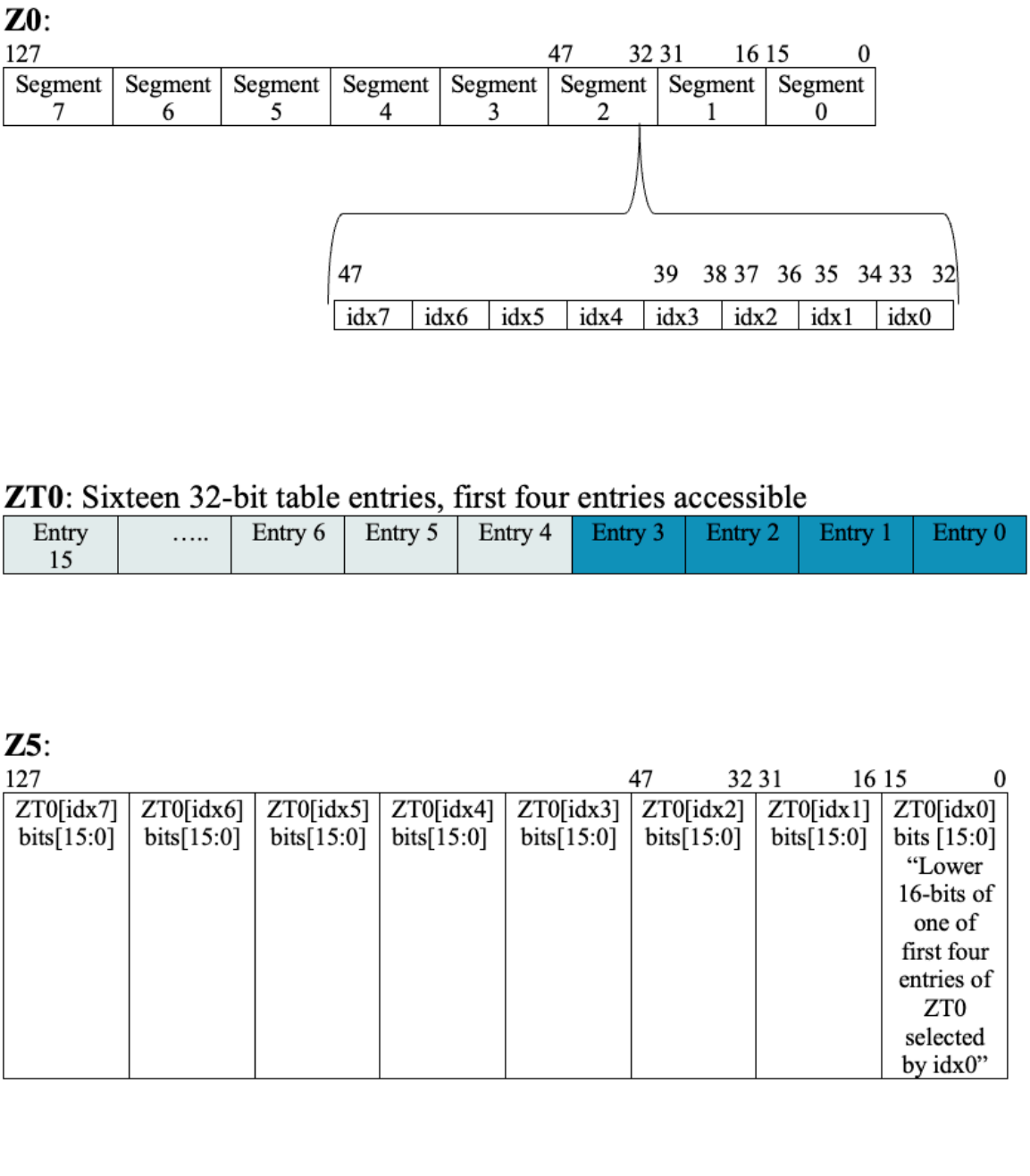
Lookup table example

The following instruction decompresses a sequence of 2-bit index values from the second segment of the Z0 register by extracting the indexed 16-bit entries from ZT0 and placing the 16-bit results into the Z5 vector:

```
- `LUTI2  Z5.H, ZT0, Z0[2]`
```

Because the index is 2 bits, only the first four entries of the lookup table can be accessed by this instruction. The element size in this instruction is 16 bits. The number of segments is determined by the element size divided by product of index size and number of registers to be loaded. For this example the number of segments is 8 and the segment size is 16 bits for an example 128-bit SVL. Eight 2-bit indices are used from the second segment of the Z0 register (as indicated in the instruction using Z0[2]) to index the first four table entries in ZT0 register. The indexed 16-bit table entries are copied to Z5 register as Figure 2-1 shows.

Figure 2-1: Example LUTI2 instrcution operation



2.1.3 SME2 multi-vector predication

The SME2 architecture introduces multi-vector instructions that operate on groups of Z vector registers and ZA array vectors. The multi-vector load and store instructions use an alternative predication mechanism to the original SVE predication mechanism, to control operations performed on a group of Z vector registers. The multi-vector predication concept is referred to as predicate-as-counter.

Unlike predication in SVE, which is a bitmask based, predicate-as-counter uses an encoded element count in the predicate register. The encoded value indicates the number of consecutive elements starting from element 0 that are Active, with the remaining elements Inactive, or vice versa.

2.2 Streaming SVE mode

An implementation of SME supports Streaming SVE mode. The Streaming SVE mode is a dedicated mode for SME operations that can be enabled or disabled by software by programming the PSTATE.SM field.

When the PE is in Streaming SVE mode:

- The streaming vector registers Z0-Z31, streaming predicate registers P0-P15 and SME architecture state are accessible by SME instructions and a subset of SVE2 instructions executable in Streaming SVE mode.
- The effective vector length changes to streaming vector length. The SMCR_EL1, Streaming SVE Mode Control Register for EL1 configures the Effective Streaming SVE vector length when the PE is in Streaming SVE mode and executing at EL1 or EL0. For EL2 and EL3, corresponding SMCR register configures the Effective Streaming SVE vector length.

SVL is independent of SVE Vector length (referred to as VL which is the vector length when not in Streaming SVE mode). The Effective Streaming SVE vector length, SVL, is a power of two in the range 128-2048 bits inclusive. SVL can vary between implementations. When streaming SVE mode is disabled, the ZCR_ELx register determines the effective SVE vector length (VL).

Streaming SVE mode is enabled when PSTATE.SM=1. When PSTATE.SM is changed from 0 to 1, Streaming SVE mode is entered and SVE registers Z0-Z31 and P0-P15 in the new mode are set to 0.

You cannot directly program the PSTATE.SM field. The PSTATE.SM field can be programmed by using the SVCR register. You can use the following instruction to independently set or clear PSTATE.SM field:

- `MSR SVCRSM, #<imm1>`

Also, you can use `SMSTART SM` instruction, alias of the `MSR SVCRSM, #1` and `SMSTOP SM` instruction, alias of the `MSR SVCRSM, #0`.

You can use the MRS instruction to read the SVCR register.

When the PSTATE.SM is changed from 1 to 0, an exit from Streaming SVE mode is performed, and each implemented bit of the SVE registers Z0-Z31 and P0-P15 in the new mode will set to zero.


Note

SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.

2.3 SME ZA storage

Streaming SVE mode permits access to the ZA array when ZA storage is enabled. When ZA storage is enabled, the ZA-targeting instructions are available.

You can enable or disable the SME ZA storage in software by programming the PSTATE.ZA field. The SME ZA storage is enabled when PSTATE.ZA is 1. When ZA storage is enabled:

- The content of ZA storage is valid and is retained by hardware irrespective of whether the PE is in Streaming SVE mode.
- If SME2 is implemented ZT0 register is enabled.

You cannot directy program the PSTATE.ZA field. The PSTATE.ZA field can be programmed using the SVCR register. You can use the following instruction to independently set or clear PSTATE.ZA field: - `MSR SVCRZA, #<imm1>`

Also, you can use the `SMSTART ZA` instruction, an alias of the `MSR SVCRZA, #1` and `SMSTOP ZA` instruction, an alias of the `MSR SVCRZA, #0`.

You can use the MRS instruction to read the SVCR register.

You can use the `SMSTART` instruction to enable both Streaming SVE mode, and SME ZA array storage. You can use the `SMSTOP` instruction to disable Streaming SVE mode, and SME ZA array storage.


Note

SMSTART ZA enables the SME ZA array storage, but does not cause an entry to Streaming SVE mode.

2.3.1 ZA array vector access and ZA tile mapping

This section describes the ZA Storage, ZA array vector access, and ZA tile mappings for 8-bit, 16-bit, and 32-bit data types in an example 128-bit SVL implementation.

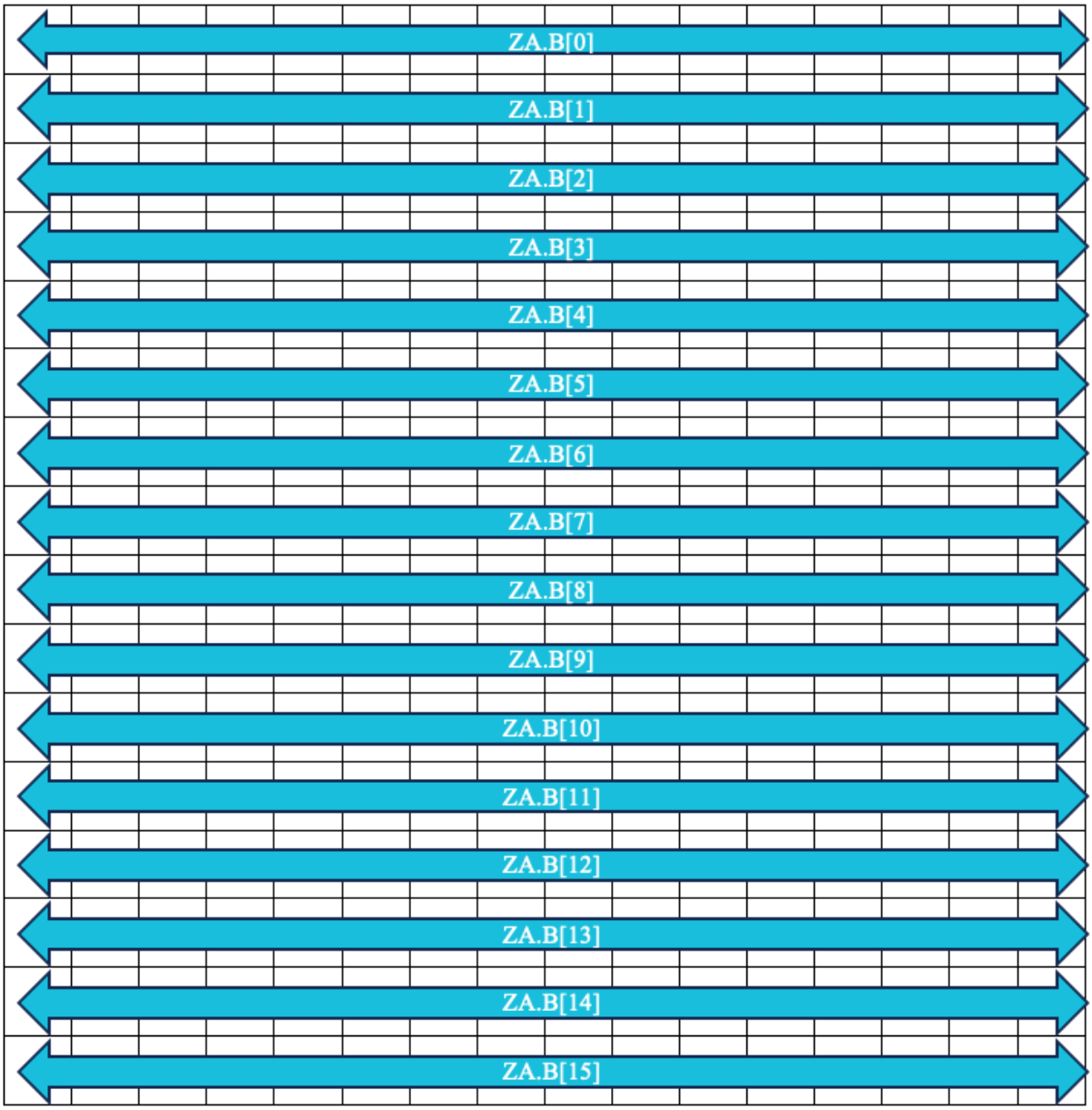
2.3.1.1 ZA storage access for 8-bit element size

Each small box in the figures of this section represents an 8-bit element.

ZA array vector access for 8-bit element size

Figure 2-2 shows ZA Storage with ZA array vector access for 8-bit element size.

Figure 2-2: ZA array vector access in ZA storage for 8-bit element size for an 128-bit SVL implementation



Accessing ZA tile for 8-bit element size

When accessing 8-bit element ZA tile, there is a single tile ZA0.B which consists of [16 × 16] 8-bit elements and occupies all of the ZA storage.

Figure 2-3 shows the horizontal slice of ZA0.B tile.

Figure 2-3: Horizontal slices in ZA0.B tile

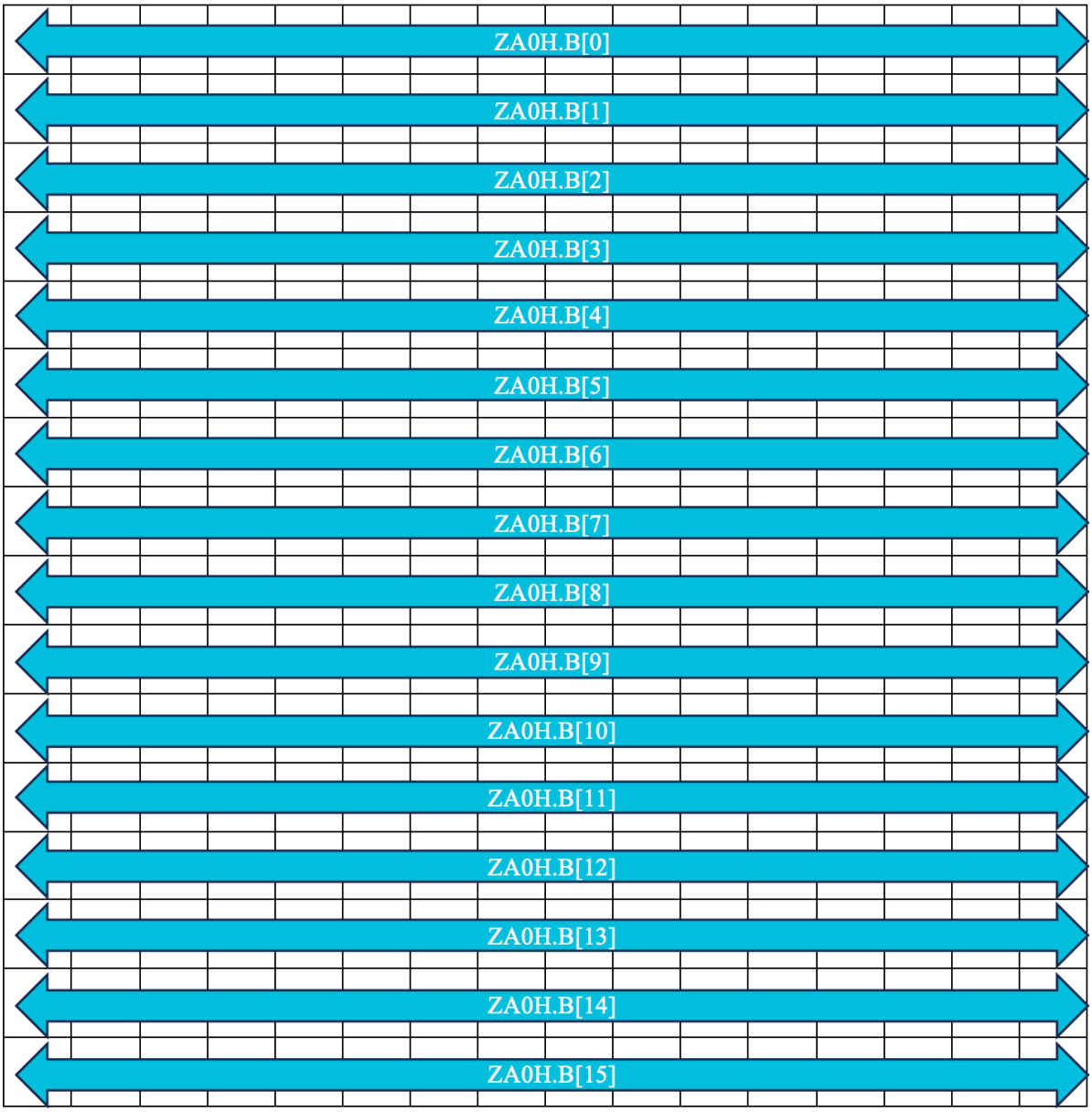


Figure 2-4 shows the vertical slice of ZA0.B tile.

Figure 2-4: Vertical slices in ZA0.B tile

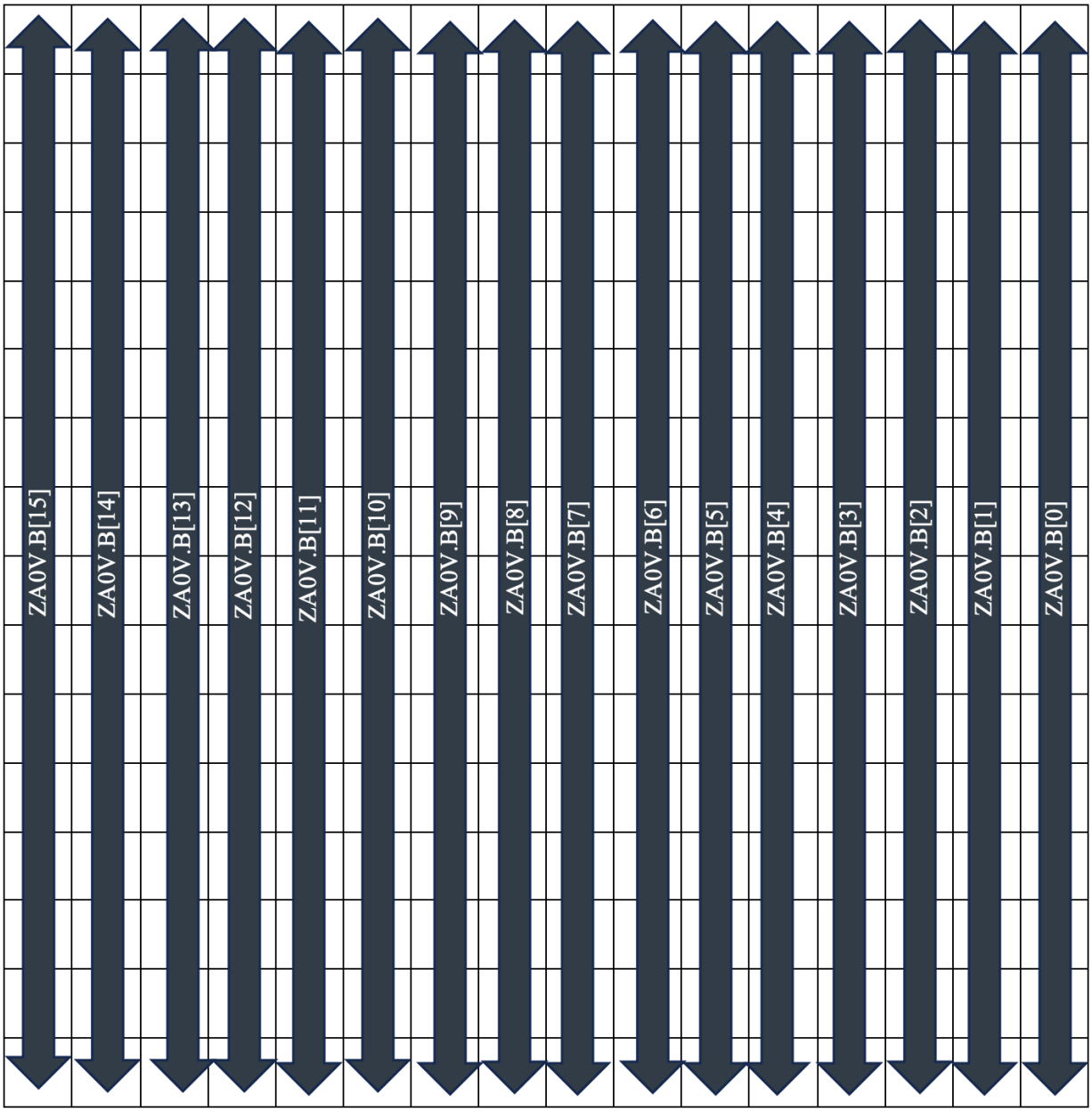


Figure 2-2, Figure 2-3, and Figure 2-4 show ZA storage being accessible as:

- Vectors of 8-bit element size
- Horizontal and vertical slices in a tile of 8-bit element size

The subsequent sections depict ZA storage being accessible as:

- Vectors of 16-bit and 32-bit element sizes
- Horizontal and vertical slices in a tile of 16-bit and 32-bit element sizes

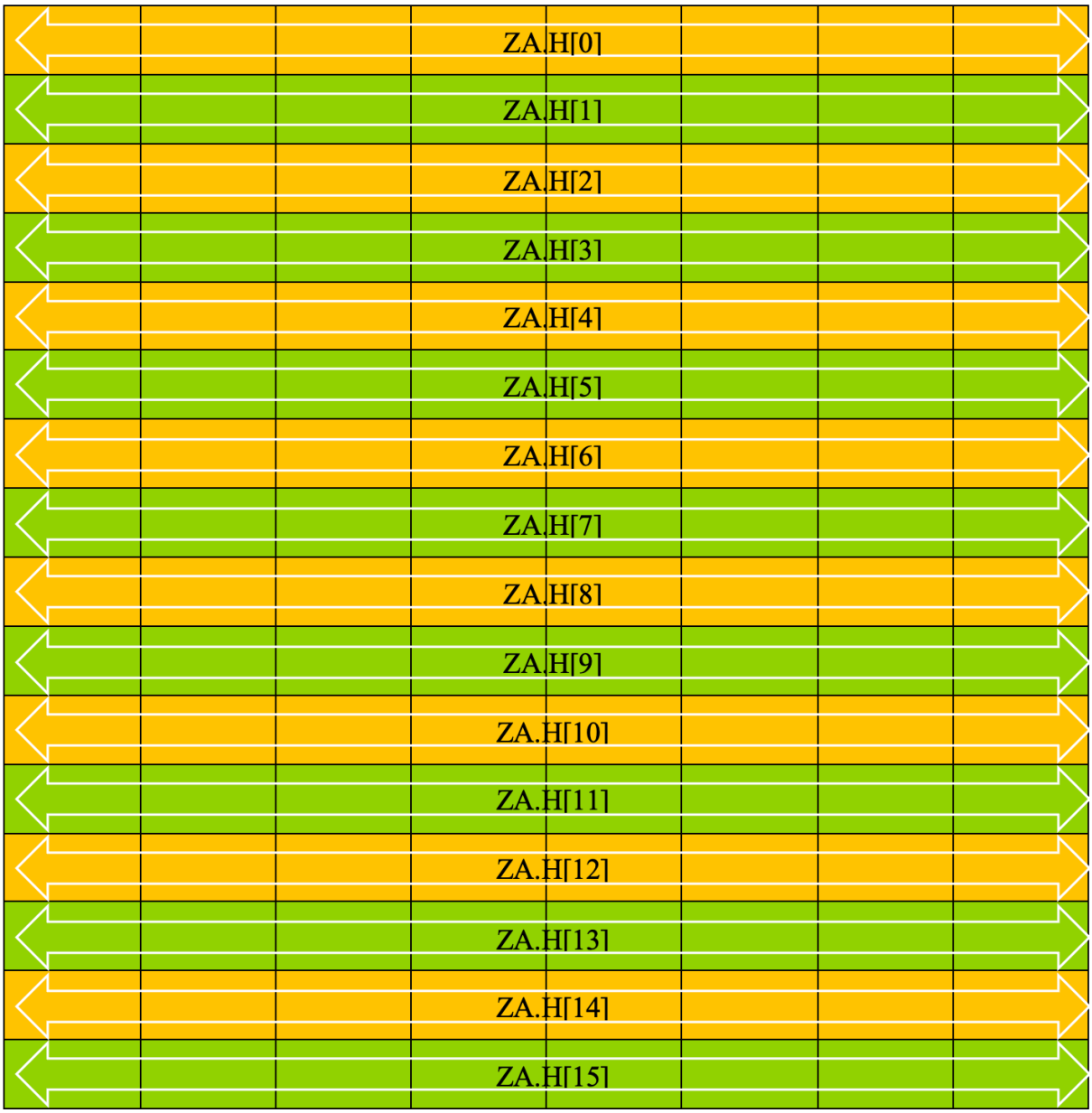
2.3.1.2 ZA storage access for 16-bit element size

Each small box in the figures of this section represents an 16-bit element.

ZA array vector access for 16-bit element size

Figure 2-5 shows ZA Storage with ZA array vector access for 16-bit element size. The color coding used in the following figure shows how the vectors are mapped to different tiles that are described in subsequent sections.

Figure 2-5: ZA array vector access in ZA Storage for 16-bit element size for an 128-bit SVL implementation



Accessing ZA tile for 16-bit element size

While accessing 16-bit element ZA tile, there are two tiles, ZA0.H and ZA1.H. Each tile consists of [8 × 8] 16-bit elements and occupies half of the ZA storage.

Figure 2-6 shows the horizontal slices of ZA0.H tile in the ZA storage.

Figure 2-6: Horizontal slices in ZA0.H tile in ZA storage

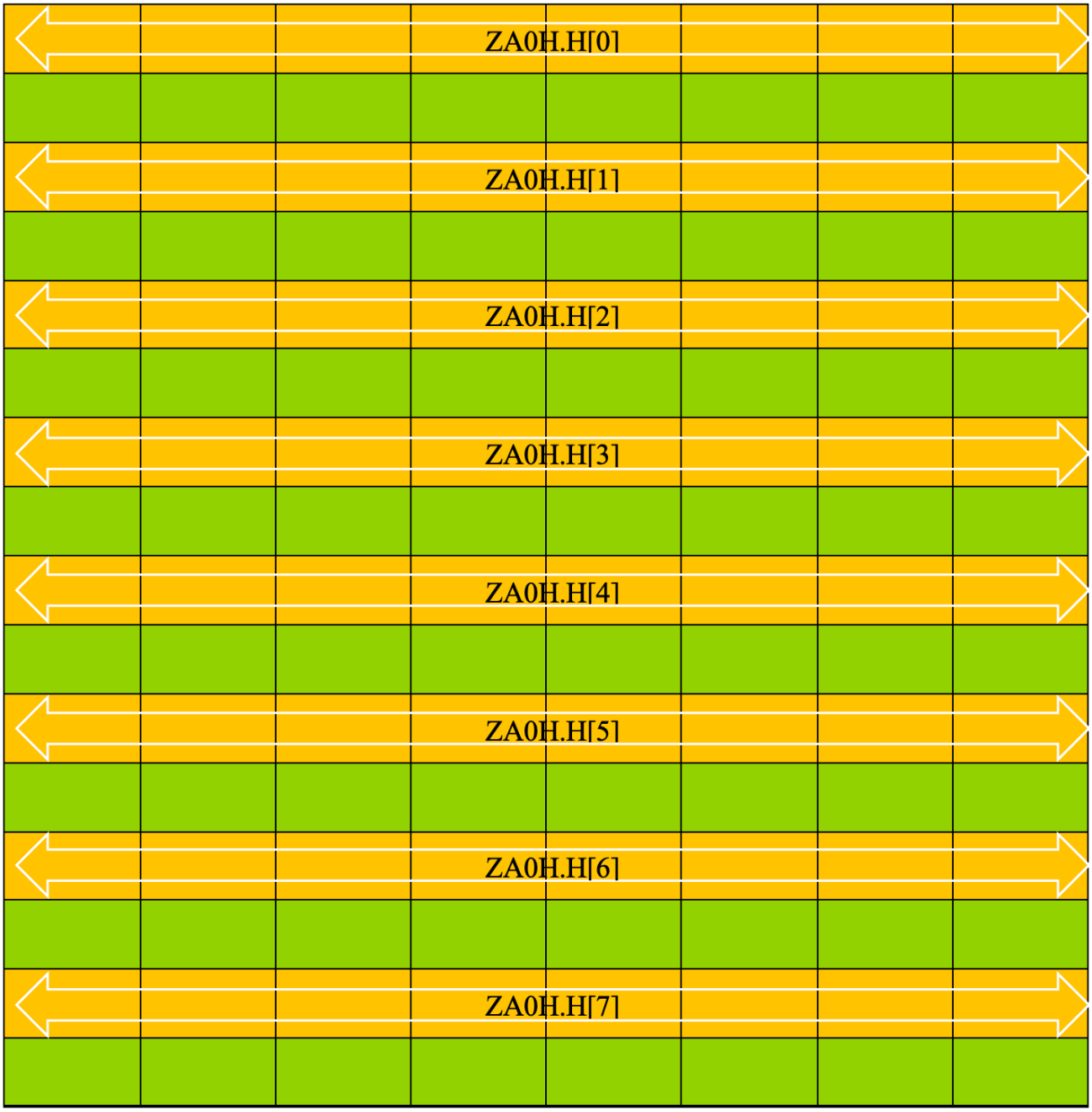


Figure 2-7 shows vertical slices in the ZA0.H tile.

Figure 2-7: Vertical slices in ZA0.H tile

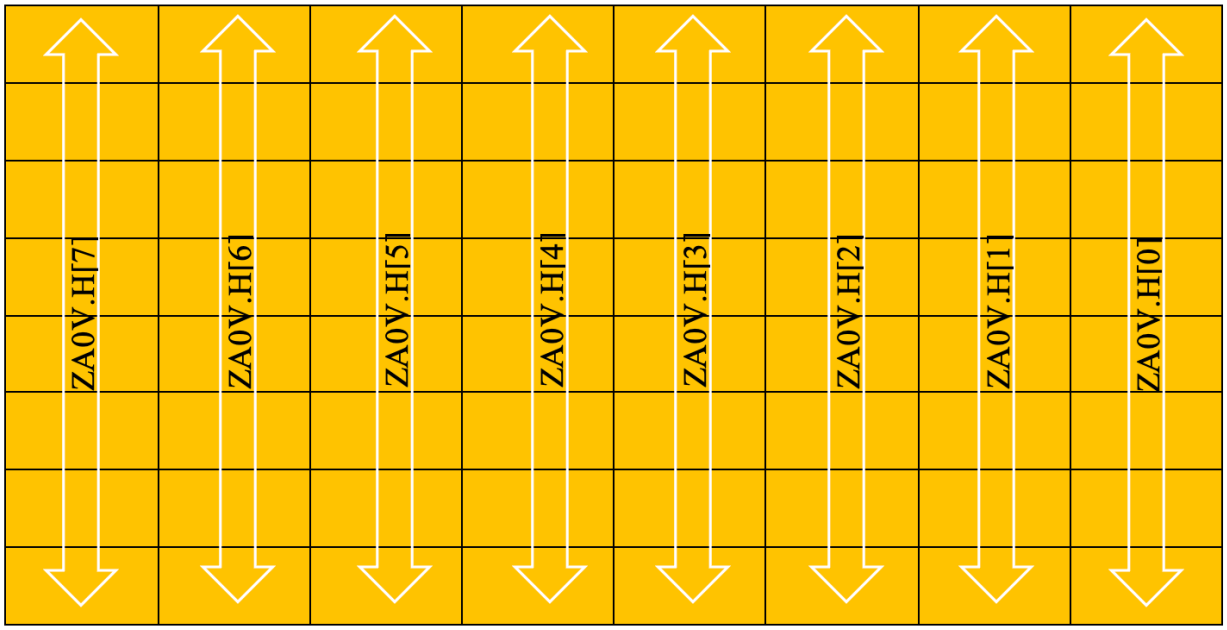


Figure 2-8 shows the horizontal slices of ZA1.H tile in the ZA storage.

Figure 2-8: Horizontal slices in ZA1.H tile in ZA Storage

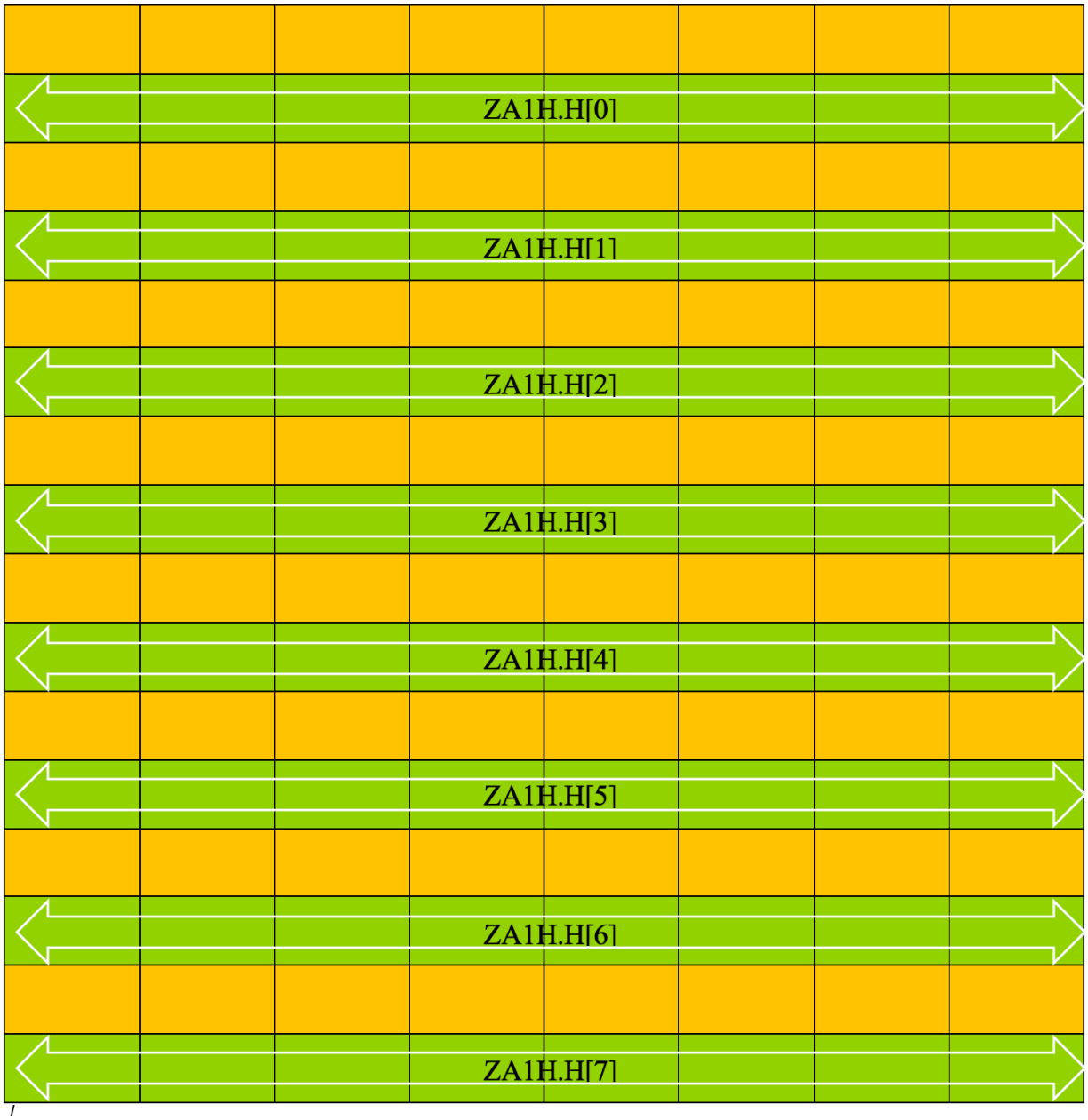


Figure 2-9 shows the horizontal slices in ZA1.H tile

Figure 2-9: Horizontal slices in ZA1.H tile

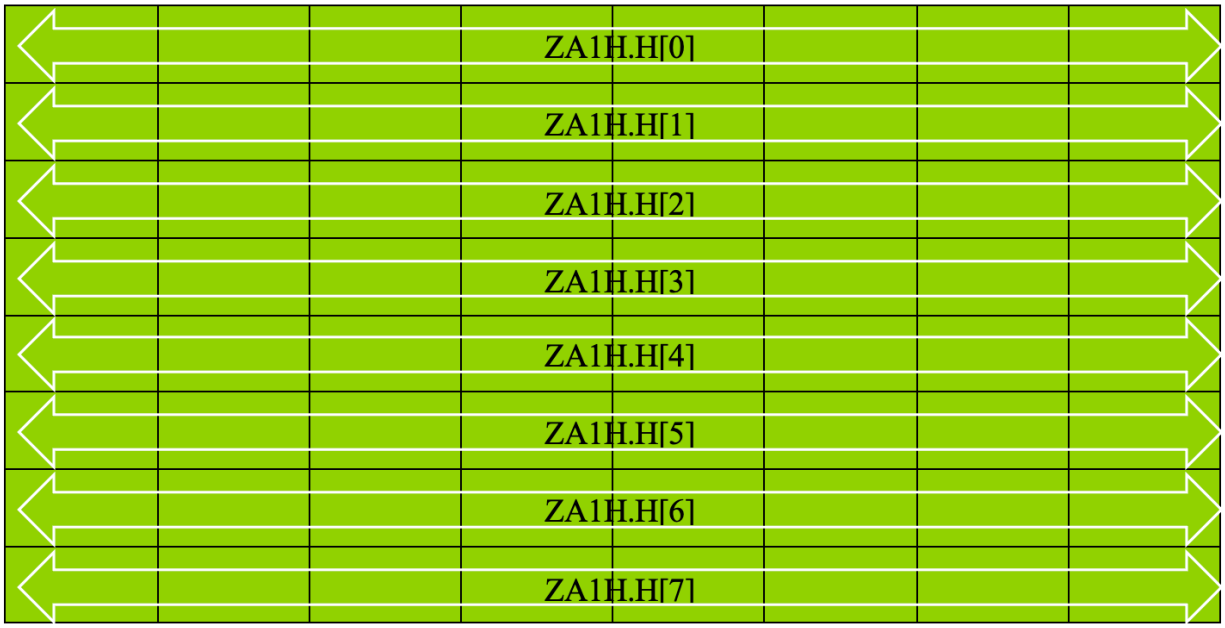
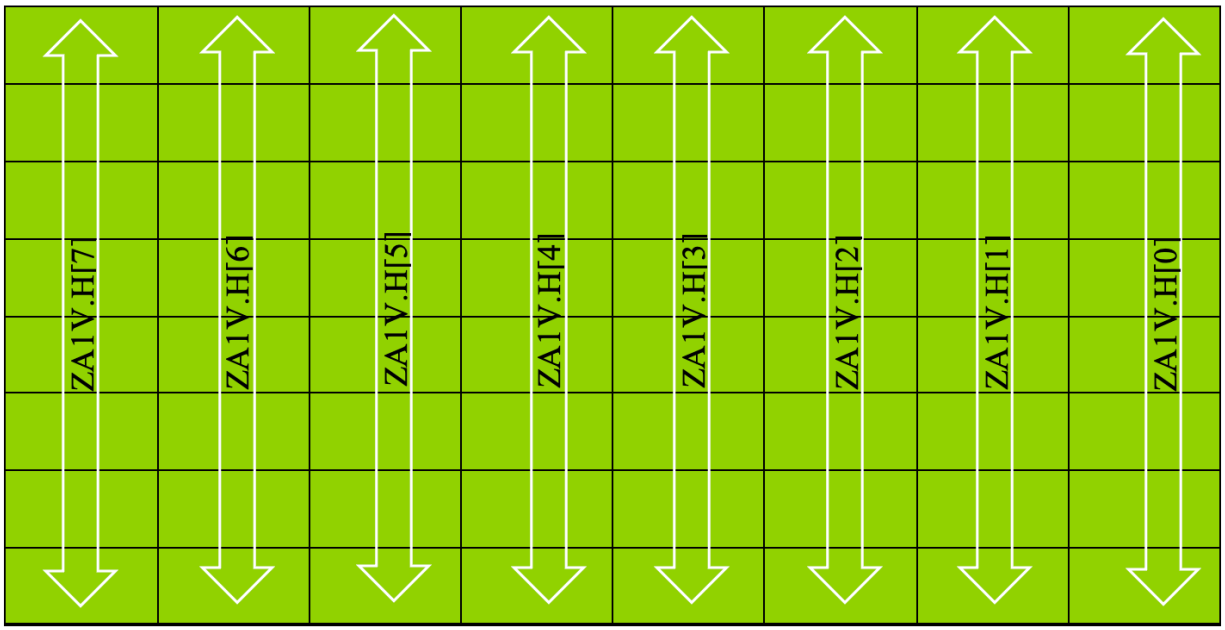


Figure 2-10 shows the vertical slices in ZA1.H tile.

Figure 2-10: Vertical slices in ZA1.H tile



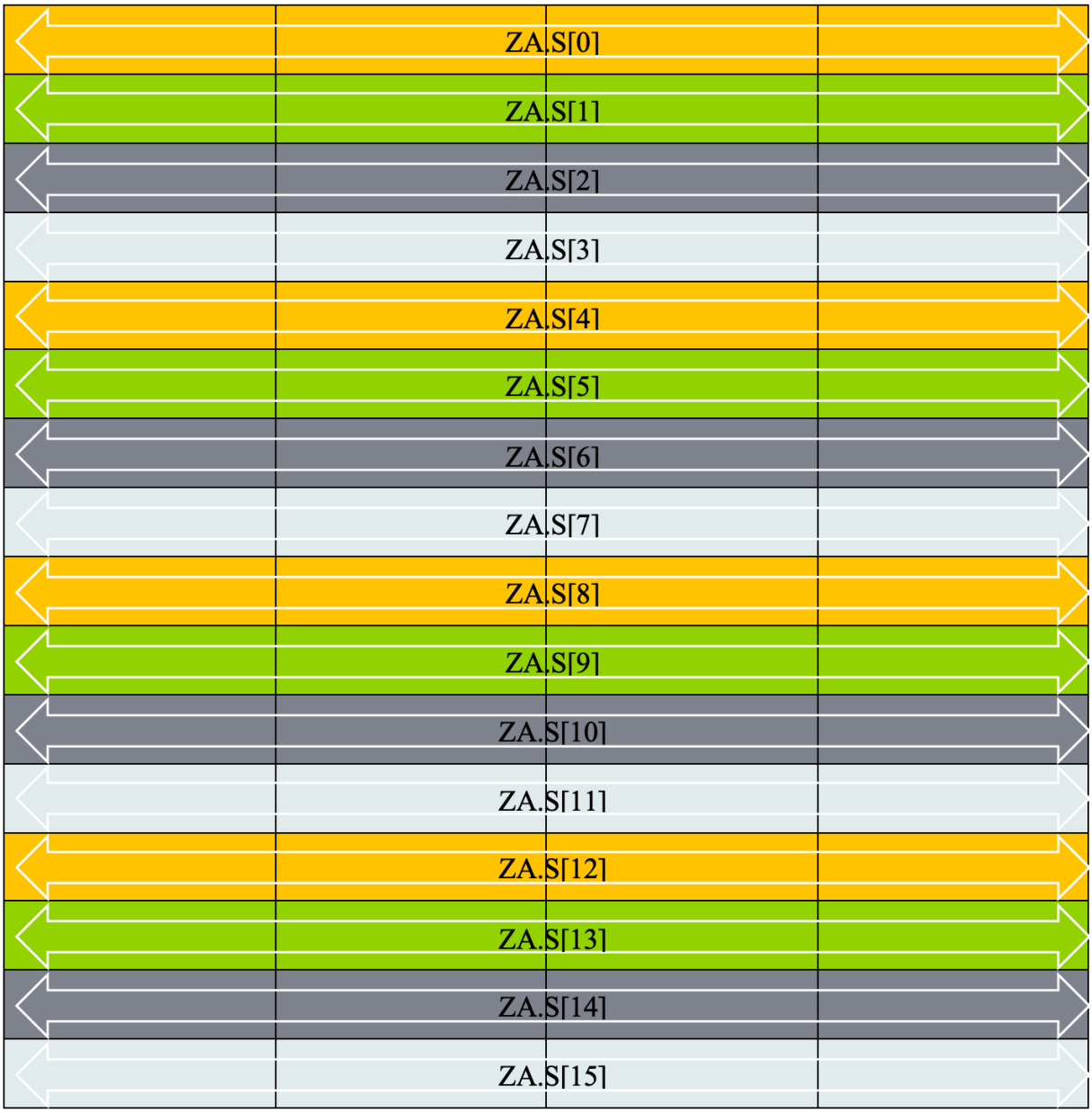
2.3.1.3 ZA storage access for 32-bit element size

Each small box in the figures of this section represents an 32-bit element.

ZA array vector access for 32-bit element size

Figure 2-11 shows ZA Storage with ZA array vector access for 32-bit element size. The color coding used in the following figure illustrates how the vectors are mapped to different tiles that are described in subsequent sections.

Figure 2-11: ZA array vector access in ZA Storage for 32-bit element size for an 128-bit SVL implementation



Accessing ZA tile for 32-bit element size

While accessing 32-bit element ZA tile, there are four tiles, ZA0.S, ZA1.S, ZA2.S, and ZA3.S. Each tile consists of [4 × 4] 32-bit elements and occupies a quarter of the ZA storage.

Figure 2-12 shows the horizontal slices of ZA0.S, ZA1.S, ZA2.S, and ZA3.S tiles in the ZA storage.

Figure 2-12: Horizontal slices in ZA0.S, ZA1.S, ZA2.S and ZA3.S tiles in ZA storage

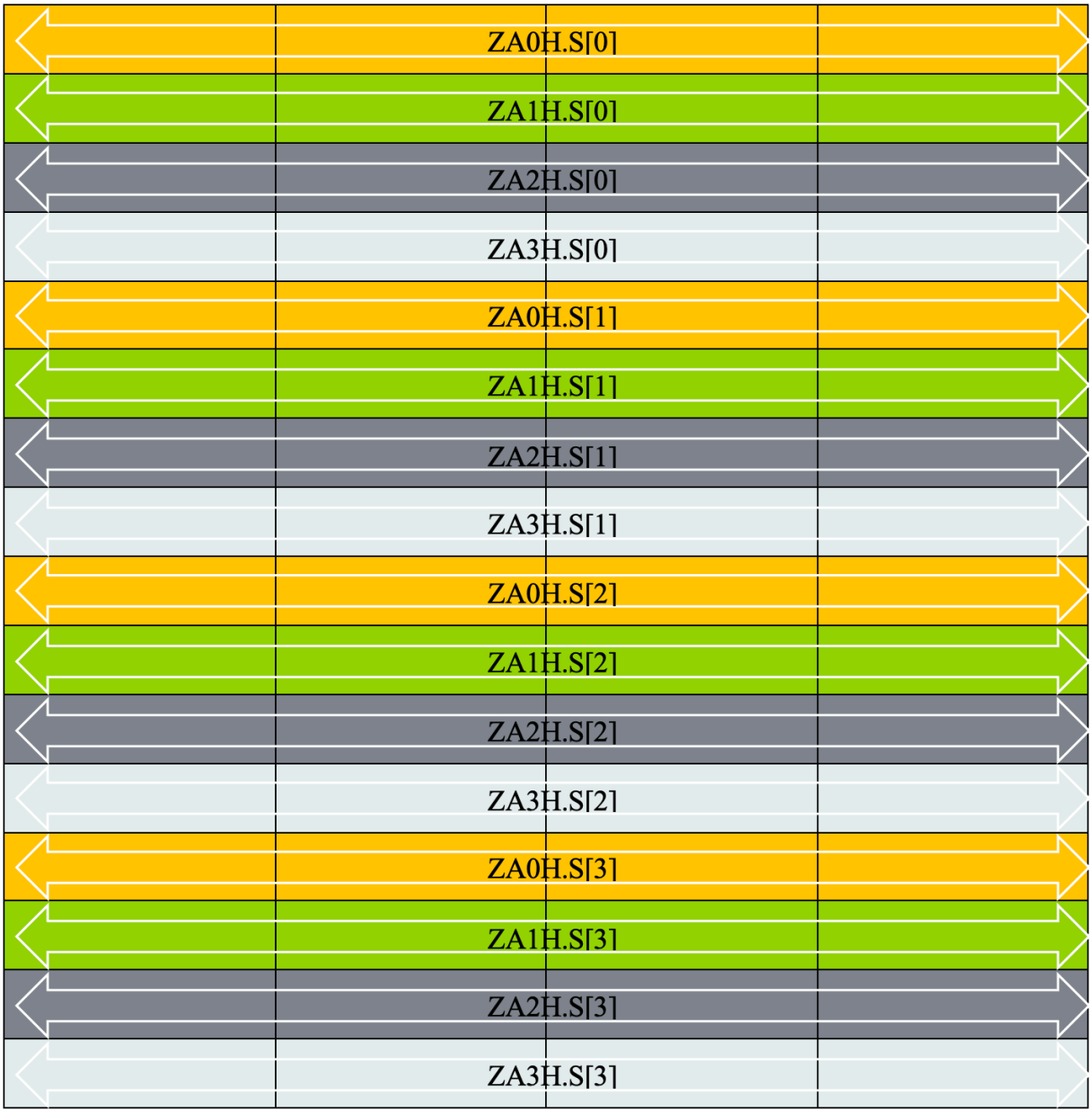
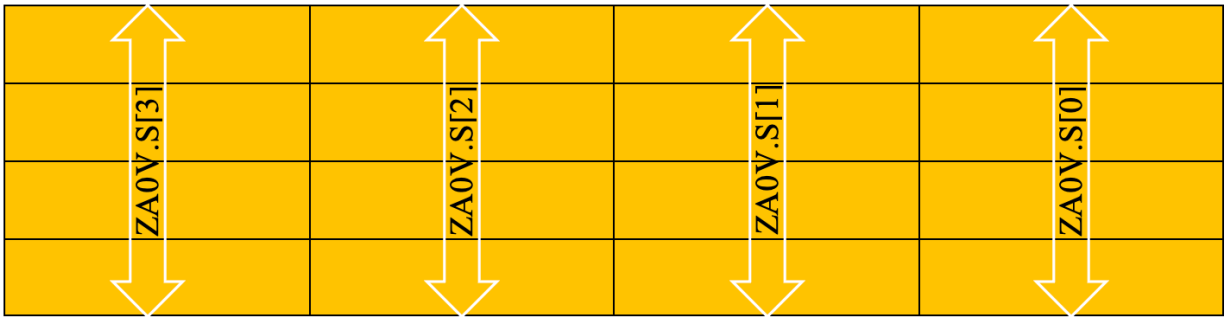


Figure 2-13 shows the vertical slices in ZA0.S.

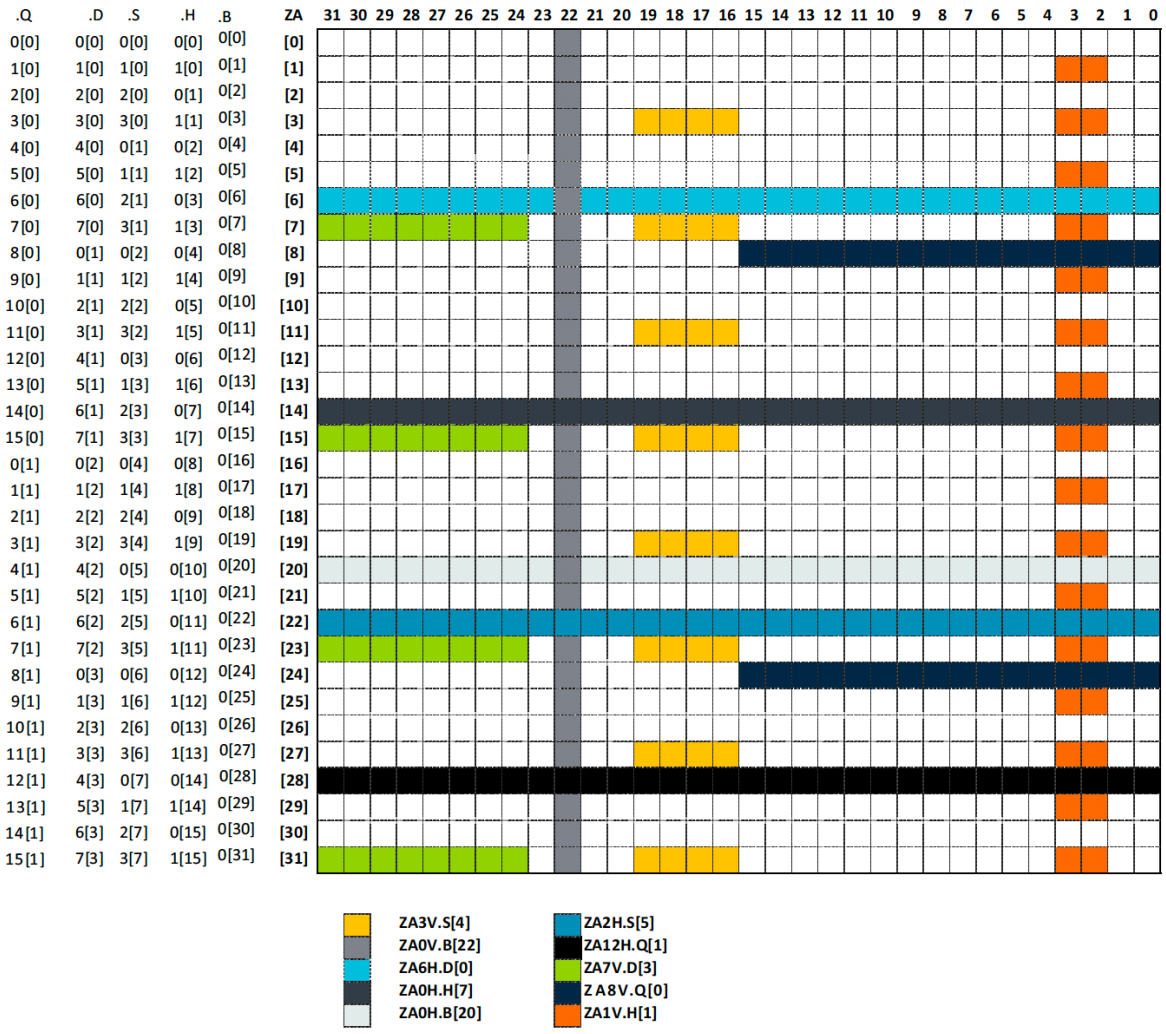
Figure 2-13: Vertical slice of ZA0.S tile



Similar to the above ZA1.S, ZA2.S, and ZA3.S tiles have ZA1V.S[0-3], ZA2V.S[0-3], ZA3V.S[0-3] vertical slices correspondingly.

The above examples show the ZA Storage mapping for an SVL of 128-bits for 32-bit, 16-bit and 8-bit element sizes. The following figure illustrates the ZA storage mapping for an SVL of 256 bits, for various element size tiles, horizontal tile slices, and vertical tile slices. Each small square represents 8 bits.

Figure 2-14: ZA Storage mapping for 256 bits SVL implementation



2.4 SME2 multi-vector operands

The SME2 adds instructions capable of supporting multi-vector operands, where the source and destination operands can be one of the following:

- A group of one, two or four SVE Z vector registers
- A group of one, two or four ZA tile slices
- A group of one, two, four, eight, or sixteen ZA array vectors

2.4.1 Z multi-vector operands

A multi-vector operand consisting of two or four SVE Z vector registers is called a Z multi-vector operand.

The Z multi-vector operand can be consecutive Z registers, or Z registers with strided numbering.

For example:

- Consecutively numbered Z register operands:
  - `FCVTZS { Z0.S-Z1.S }, { Z4.S-Z5.S }`, converts each element of Z4 and Z5 to the signed 32-bit integer nearer to zero from single-precision, and places the results in the corresponding elements of the Z0 and Z1.
- Z register operands with strided numbering:
  - `LD1D { Z0.D, Z4.D, Z8.D, Z12.D }, P8/Z, [X0]`, contiguously loads unsigned doublewords into the elements of vector registers Z0, Z4, Z8, and Z12 from the memory pointed by X0.

2.4.2 ZA multi-slice operands

A multi-vector operand consisting of two or four ZA tile slices is called a ZA multi-slice operand. A ZA multi-slice operand can be consecutively numbered horizontal ZA tile slices or consecutively numbered vertical ZA tile slices.

For example:

`MOVA ZA0H.B [W12, 0:3], { Z0.B-Z3.B }`, copies Z0 to Z3 vector contents into the four horizontal slices ZA0H.B[5] to ZA0H.B[8], assuming W12 value is 5.

2.4.3 ZA multi-vector operands

The ZA multi-vector operand consists of one, two, or four vector groups, where a vector group is one of the following:

- ZA single-vector group, any one ZA array vector
- ZA double-vector group, two consecutively numbered vectors in the ZA array
- ZA quad-vector group, four consecutively numbered vectors in the ZA array

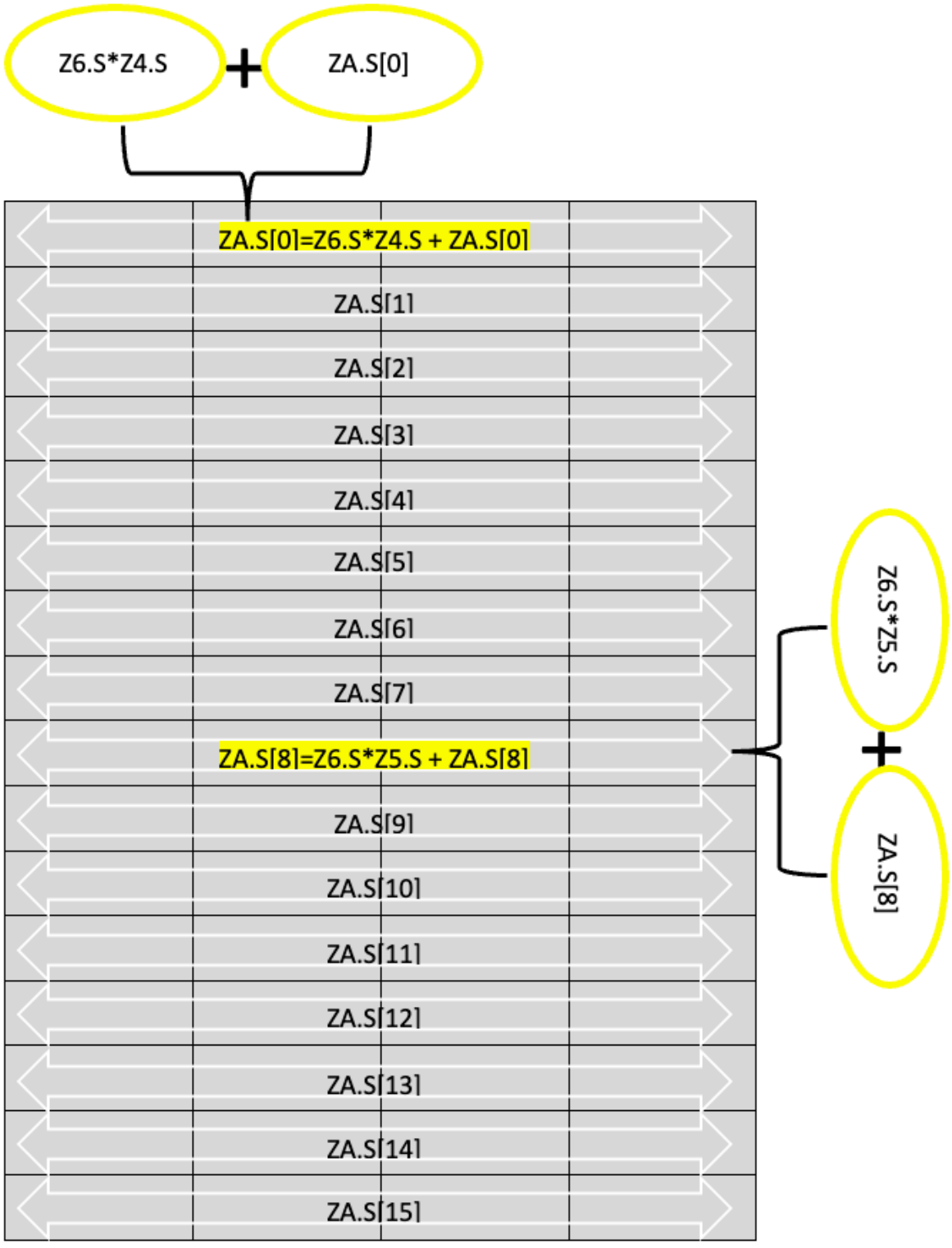
The vector group symbols, VGx2 and VGx4, are used to indicate two or four vector groups respectively.

Example instruction that produces results to two ZA single-vector groups ZA.S[0] and ZA.S[8]

`FMLA ZA.S[W8, 0, VGx2], {Z4.S-Z5.S}, Z6.S`, multiplies each 32-bit floating point element of Z6 with corresponding 32-bit floating point elements of Z4 and Z5 registers, and add the results

to corresponding 32-bit elements in ZA.S[0] and ZA.S[0+SVLb/2], assuming W8 value is 0. Figure 2-15 shows the FMLA instruction operation for an example 128-bit SVL.

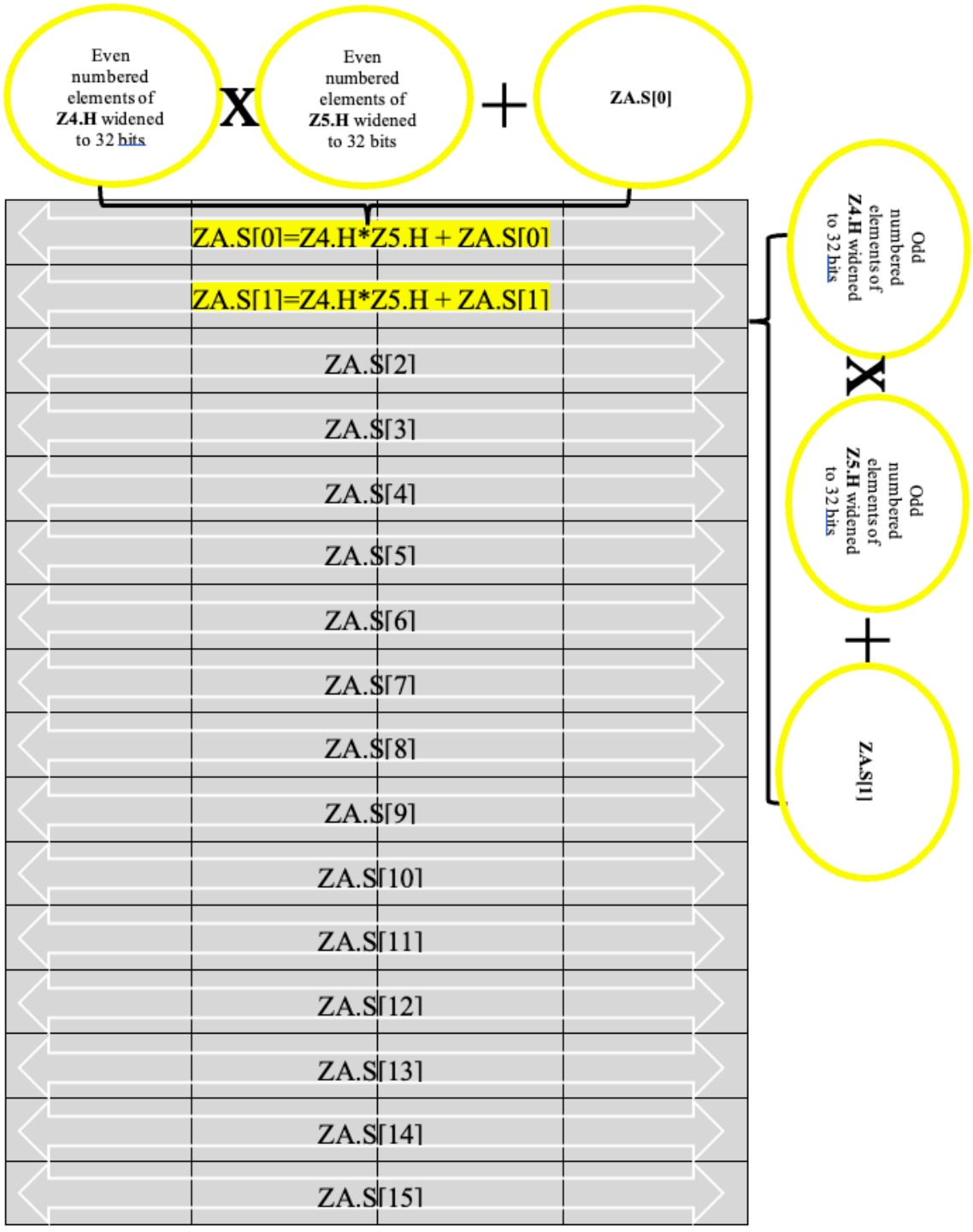
Figure 2-15: FMLA instrcution operation



Example instruction that produces results to one ZA double-vector group that consists ZA.S[0] and ZA.S[1]

`FMLAL ZA.S[W8, 0:1], Z4.H, Z5.H`, widens all 16-bit half-precision elements in Z4 and Z5, and multiplies the corresponding elements, producing SVLb/2 number of 32-bit intermediate results. The even numbered intermediate results are added to 32-bit elements in ZA.S[0] and the odd numbered intermediate results are added to 32-bit elements in ZA.S[1], assuming W8 value is 0. Figure 2-16 shows the FMLAL instruction operation for an example 128-bit SVL.

Figure 2-16: FMLAL instrcution operation



Example instruction that produces results to two ZA quad-vector groups ZA.S[0-3] and ZA.S[8-11]

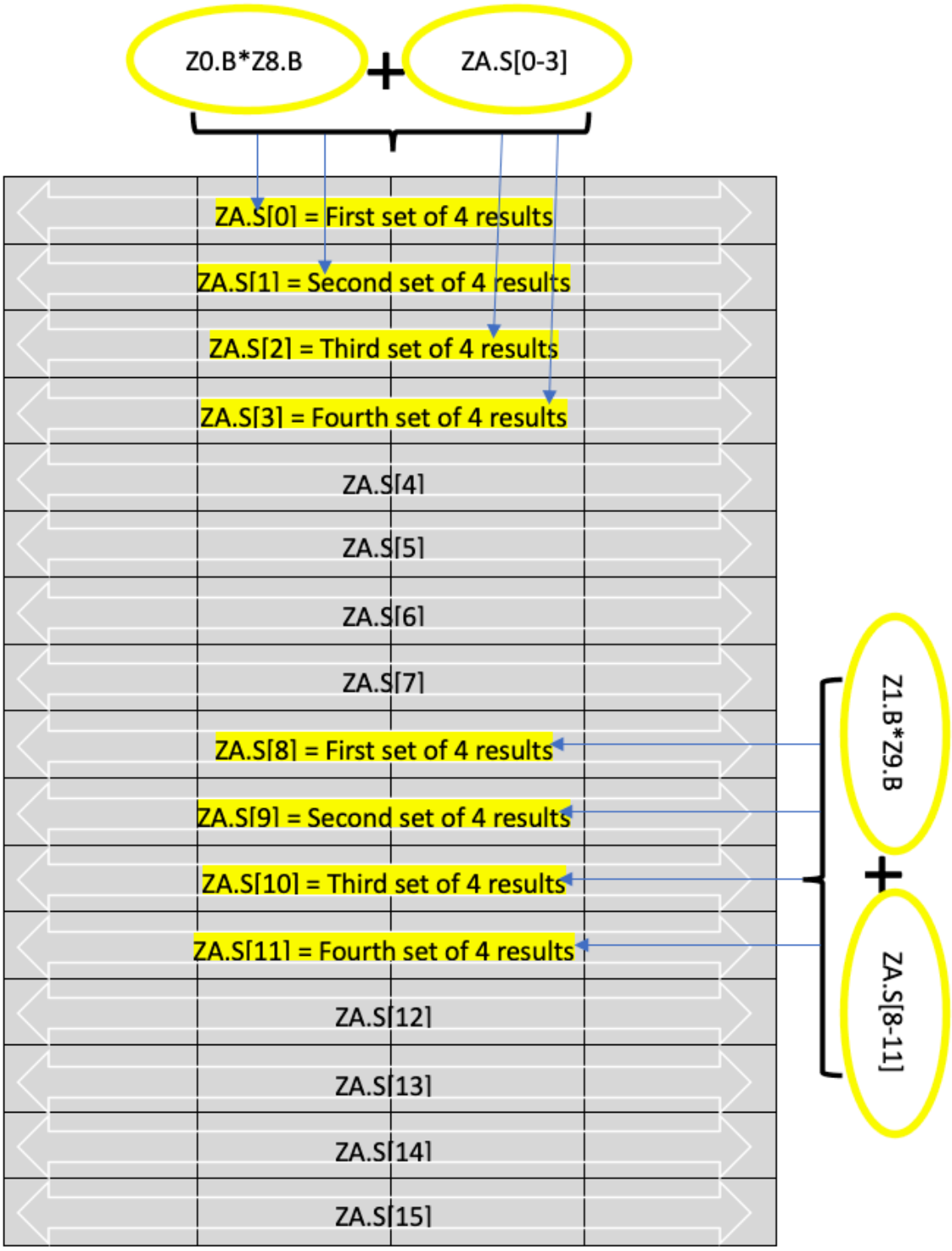
`USMLALL ZA.S[W8, 0:3, VGx2], {Z0.B-Z1.B}, {Z8.B-Z9.B}`, multiplies each unsigned 8-bit element in the Z0.B and Z1.B source vectors with each signed 8-bit element in the Z8.B and

Z9.B second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of two ZA quad-vector groups ZA.S[0] to ZA.S[3] and ZA.S[SVLb/2+0] to SVLb/2+3], assuming W8 value is 0. Figure 2-17 shows the USMLALL instruction operation for an example 128-bit SVL.

In Figure 2-17,

- The first set of four results are formed by elements i*4, where i is 0,1,2,3.
- The second set of four results are formed by elements (i*4)+1, where i is 0,1,2,3.
- The third set of four results are formed by elements (i*4)+2, where i is 0,1,2,3.
- The fourth set of four results are formed by elements (i*4)+3, where i is 0,1,2,3.

Figure 2-17: USMLALL instrcution operation



2.5 SME context save restore

This section describes SME context save and restore operations.

2.5.1 Context save restore on entry or exit from Streaming SVE mode

On entry to Streaming SVE mode and on exit from Streaming SVE mode, each implemented bit of SVE registers Z0-Z31 and P0-P15 is set to 0. In addition FPSR is set to a fixed value. Software must save these registers before entering Streaming SVE mode and restore these registers after exiting Streaming SVE mode.

2.5.2 Context save restore in supervisory software

To avoid data leakage between software bodies of different trust/security scope, supervisory software must ensure context save restore for ZA Storage and ZT0 register when performing context switch between the software bodies.

Supervisory software must not assume a lower value of SVL being used by the software bodies it manages. The supervisory software must also save restore ZA storage that is being exposed to the software bodies.

3. Toolchains and model support

This chapter describes the software tools and execution model that support SME/SME2 application development.

3.1 Quick start example for SME/SME2

This section shows a quick start example based SME/SME2 assembler instructions and how to run it on Arm Development Studio. It also describes the source code, compiling commands with Arm compiler 6 and execution on the FVP model.

Download and install the Arm Development Studio by referring the Arm Development Studio Getting Started Guide.

3.1.1 Step 1: Create a new project with SME/SME2 instruction

In the Arm Development Studio menus,

1. Click **File -> New->Project -> C/C++ -> C/C++ project**;
2. Choose **C Manage build or C++ -> Empty Project with Arm Compiler for Embedded 6** (type the custom project name);
3. Add the main source file in this project with the following code:

```
// SME_example.s
<...>
// <Definition of Initializing sequence about preparation to enter the streaming
 mode>
.macro sme_init
...

// <Definition of Initializing sequence about preparation to exit the streaming
 mode>
.macro sme_done
...

sme_funcs1:
    sme_init
    mov     w12,  #0x0
    mov     x13,   #0x80000000
    ptrue   pn8.b
    // Contiguous load from memory address stored at x13 to Z registers
    ld1b    {z0.b-z3.b}, pn8/z, [x13]
    // Move the value from Z registers to ZA tiles
    mova    za0h.b[w12, 0:3], {z0.b-z3.b}
    sme_done
<...>

// main.c
<...>
```

```
#include <stdio.h>

extern sme_funcs1();

int main() {
    sme_funcs1();
    return 0;
}
<...>
```

The code shows a simple process that enables the streaming SVE mode and loads values to the ZA register by inserting the SME/SME2 assembler instruction. Figure 3-1 shows the content of ZA storage before executing the `mova` instruction.

Figure 3-1: Chart for ZA0 content before executing mova instruction

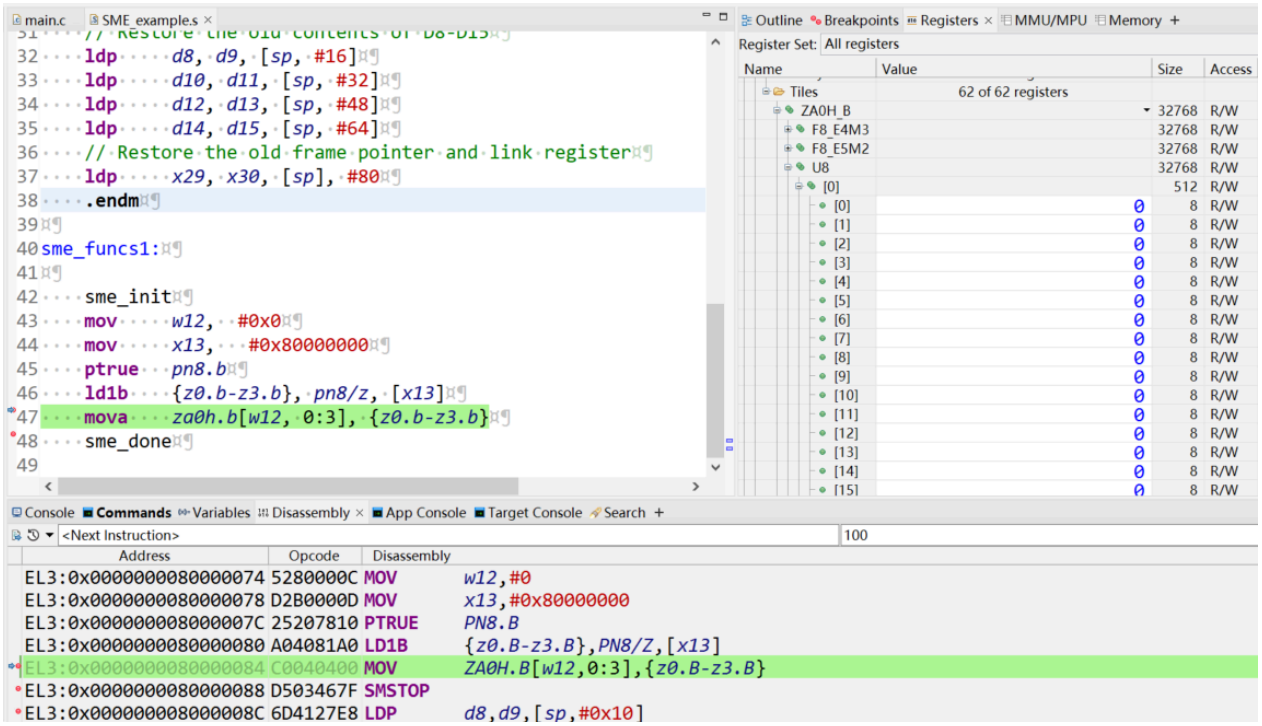
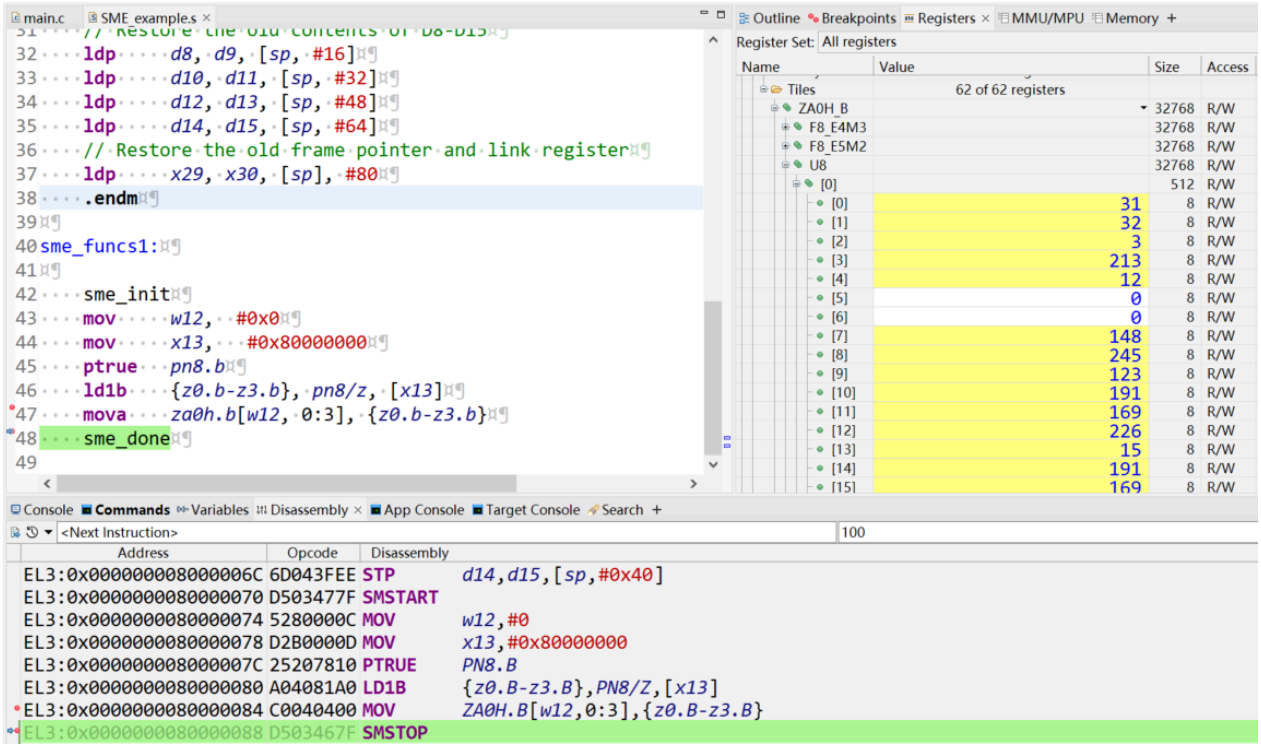


Figure 3-2 shows the content change of ZA0 tile through setting breakpoints after executing the `mova` instruction.

Figure 3-2: Chart for ZA0 content after executing mova instruction



Note

See the details of macros `sme_init` and `sme_done` in Calling conventions.

3.1.2 Step 2: Build the project

In the Arm Development Studio menus,

1. Right-click the project and select properties;
2. Open the **C/C++ Build -> Setting -> Tool Settings**;
3. For all Tools Settings, select the **Generic Armv9.2-A AArch64** as Target CPU, **Armv8 (Neon)** as target FPU;
4. Build the project with the below setting.

```
<...>
// Arm C Compiler for Embedded 6 setting is below
--target=aarch64-arm-none-eabi -march=armv9.2-a+sme2 -O1 -g -fno-inline-functions

// Arm assembler 6 setting is below
--target=aarch64-arm-none-eabi -march=armv9.2-a+sme2 -g

// Arm Linker 6 setting is below
--ro_base=0x80000000 --info=sizes
```

```
<...>
```


Note

This is the basic setting of tools with SME2 enablement. See more features in Compiler options and pragmas.

3.1.3 Step 3: Connect a Debugger and configure

In the Arm Development Studio menus,

1. Right-click the project;
2. Select the debug configuration and create a new configuration;
3. In the connection selection box, select the **FVP - Arm FVP (Installed with Arm DS) / Base_AEMvA*1_SVE/Bare Metal Debug/AEM-A_MP** with the following model parameters:

```
<...>
-C cluster0.NUM_CORES=0x1
-C bp.secure_memory=0
-C SVE.ScalableVectorExtension.has_sme=1
-C SVE.ScalableVectorExtension.has_sme2=1
<...>
```


Note

Add the following options manually: `-C bp.secure_memory=0 -C SVE.ScalableVectorExtension.has_sme=1 -C SVE.ScalableVectorExtension.has_sme2=1`

1. Load the executable file into the Files setting;
2. Select the **Debug from main** at Debugger setting;
3. Click debug, to build a Iris-compliant connection between the debugger and a FVP successfully;
4. Choose the **Steping By Instruction** debug mode, that allow viewing of the changes in the ZA register.

3.2 Compiler support

As section 3.1 shows, to build an application that uses SME and SME2 instructions, you must choose a compiler that supports these instructions. The following compilers support SME and SME2:

- Arm Compiler for Embedded 6 is a cross-platform compiler for bare-metal application development. Arm Compiler 6 first introduces support for SME from version 6.17. Then it adds

additional SME and SME2 functionality in subsequent releases. Arm compiler 6 versions greater than 6.21 support SME and SME2. Arm recommends this or greater are used.

- Clang is an open-source C compiler based on LLVM. Clang version 18 and higher support SME and SME2. From version 18, Clang can compile source code containing SME and SME2 ACLE intrinsics which implement the SME and SME2 ACLE.
- GNU toolchain is a broad collection of programming tools. GNU toolchain, GCC version 14 starts to support the SME and SME2 ACLE intrinsics.

Whichever compiler you use, Arm recommends using the most recent version.

Both the Arm Compiler 6 and Clang enable:

- Assembly of source code containing SME and SME2 instructions
- Disassembly of ELF object files containing SME and SME2 instructions

For Clang, the compiler integrates the assembler, which is not the same as GCC. The GNU toolchain uses a separate package called binutils to provide the assembler, linker and associated utilities to assemble and link. For assembly of SME/SME2 instructions, the minimal required GNU binutils version is 2.41.

3.2.1 Compiler options and pragmas

The following two ways describe how to write the source code and how to compile the code with the above compilers.

- Writing assembly code is supported directly by the compiler. Both the Arm compiler and Clang are capable of generating an executable file for SME/SME2 assembly code.
- The Arm C Language Extension (ACLE) makes the SME/SME2 instructions available as intrinsic functions, which are supported by the compilers that replace the functions with corresponding instructions. It also provides facilities for managing streaming mode and ZA storage. For details about these extensions, see sme-language-extensions-and-intrinsics in ACLE. Now, only the Clang compiler can compile the ACLE code; when GCC 14 is released, it can also compile ACLE code.

When compiling the SME/SME2 instructions, the following settings are available:

1. Compiler options:

- To target SME/SME2, select the following `-march` features with different variant types.

The following table shows part of features to target SME/SME2.

Feature identifier	Feature Description	-march/-mcpu <feature> options
FEAT_SME	Scalable Matrix Extension	sme
FEAT_SME2	SME2	sme2
FEAT_SME_F64F64	SME with the double-precision variant	sme-f64f64
FEAT_SME_I16I64	SME with 16-bit integer variant	sme-i16i64

Feature identifier	Feature Description	-march/-mcpu <feature> options
...	...	...

SME is supported in Armv9.2-A architecture. There are more options, which availability and support level differ in every compiler release. See the Arm Compiler Reference Guide for the full set of supported options in each compiler release.

There is an example to compile the SME2 instruction in C language.

```
// Arm C/C++ compiler:
armclang --target=aarch64-arm-none-eabi -march=armv9.2-a+sme2 -o <Executable file>
 <source file>

// Clang compiler:
clang -target aarch64-none-elf -march=armv9.2-a+sme2 <Source files> -o <Executable
 files>
```

1. Include the relevant SME/SME2 header files and specify which target executes the code.

```
#ifdef __ARM_FEATURE_SME
#include <arm_sme.h>
#endif
```

`<arm_sme.h>` declares functions and defines intrinsics for SME and its extensions, also including `<arm_sve.h>`.

Command line examples to compile the SME/SME2 intrinsics:

```
// Clang compiler:
clang -target aarch64-none-elf -march=armv9.2-a+sme <Source files> -o <Executable
 files>
```

Passing `-march=armv9.2-a+sme` defines the macro `__ARM_FEATURE_SME`.

Currently, GCC 14 which is the latest version to support the ACLE intrinsics for SME, has not been released to public. The Clang compiles all examples with SME intrinsics instead.

3.3 Calling conventions

To program in assembly, be aware of the latest updates to the Procedure Call Standard for the Arm 64-bit Architecture (AAPCS64) within the Application Binary Interface for the Arm Architecture

(ABI) document with SME support from release 2022Q3. Arm recommends that you use the latest version.

The following example shows the AAPCS process and how the SME assembler function is called and the related parameters are passed.

```
// main.c
<...>
int main(){
  // Initilization of two input matrices, one output matrix
  int M, N, K;
  ...

  uint16_t * matLeft         = (uint16_t *) malloc(M*K*sizeof(uint16_t));
  uint16_t * matRight        = (uint16_t *) malloc(K*N*sizeof(uint16_t));
  uint64_t * matResult       = (uint64_t *) malloc(M*N*sizeof(uint64_t));
  ...

  matmul(M, K, N, matLeft, matRight, matResult);
}
<...>

// matmul.s
<...>
.text
.global matmul
.type matmul, %function

// <Definition of Initializing sequence about preparation to enter the streaming
 mode>
.macro sme_init
...

// <Definition of Initializing sequence about preparation to exit the streaming
 mode>
.macro sme_done
...

matmul:
    // matmul(M, K, N, matLeft, matRight, matResult);
    // x0 : M
    // x1 : K
    // x2 : N
    // x3 : &matLeft
    // x4 : &matRight
    // x5 : &matResult

  sme_init
  <The process of matrix-matrix multiplication>
  sme_done
<...>
```

According to the AAPCS64, the first 8 registers, r0-r7, pass argument values into a subroutine. In this case, the x0-x5 pass the value of column and row for input matrices, and the pointer for matrices. Next, use this information in the calculation of the matrix by matrix multiplication. Registers r0-r7 also return the results from a function.

In addition to r0-r7, the AAPCS64 also defines the use of other general-purpose registers and summarizes the use of general-purpose registers in the procedure call standard.

Also, SME defines several pieces of processor state: `ZA storage`, `PSTATE.SM`, `PSTATE.ZA`, and `TPIDR2_EL0`. AAPCS64 describes the management of processor state across function boundaries.


Note

Preparation for entering and exiting streaming mode shows the sequence preparation on entry or exit from streaming mode.

3.3.1 Preparation for entering and exiting streaming mode

There are some preparation steps when entering streaming mode. The instruction syntax uses qualified FP/SIMD/SVE register names. For instance, B presents the low byte, Q presents a 128-bit quadword (Neon), and Z presents a full SVE register. The compiler needs to save the content of FP/SIMD/SVE registers D8-D15 before entering streaming mode, because the AAPCS64 requires that the low 64 bits of them are not changed by the call.

For ZA storage, AAPCS64 uses a cooperative `Lazy saving` scheme after entering streaming mode. Functions that want to use ZA must check whether ZA already contains live contents first. Then they must save those contents to a provided buffer if so. The owner of the saved contents can restore them later. See details of committing-a-lazy-save in AAPCS64.

On exit from streaming mode, restore the old contents of D8-D15.

The following example shows `sme_init` and `sme_done`, the macro definitions of preparation steps.

```
<...>
  .macro sme_init
    // Store the frame pointer and link register to create a frame chain
    stp     x29, x30, [sp, #-80]!
    // Point the frame pointer to the new frame chain
    mov     x29, sp
    // Save the current contents of D8-D15, which must be preserved by the function
    stp     d8, d9, [sp, #16]
    stp     d10, d11, [sp, #32]
    stp     d12, d13, [sp, #48]
    stp     d14, d15, [sp, #64]
    // Enable streaming mode and ZA
    smstart
    // Check whether there are already some lazily-saved contents in ZA
    mrs     x0, tpidr2_el0
    cbz     x0, 1f
    // Commit the lazy save
    bl      __arm_tpidr2_save
    // Indicate that there is no longer an active lazy save
    msr     tpidr2_el0, xzr
1:
    .endm


    .macro sme_done
    // Disable ZA and exit streaming mode
    smstop
    // Restore the old contents of D8-D15
    ldp     d8, d9, [sp, #16]
    ldp     d10, d11, [sp, #32]
    ldp     d12, d13, [sp, #48]
    ldp     d14, d15, [sp, #64]
    // Restore the old frame pointer and link register
    ldp     x29, x30, [sp], #80
    .endm
```

```
<...>
```


Note

If an algorithm depends on ZA starting out at zero, it should clear out ZA after the call to `__arm_tpidr2_save`. Otherwise, the callee would inherit ZA contents from the caller. The instruction to zero ZA is `zero {za}`. If ZA was previously off, executing `SMSTART ZA` has the effect of zeroing the contents of ZA.

`__arm_tpidr2_save` is not something that C or C++ code should call. Instead, the compiler emits a call to `__arm_tpidr2_save` where it is necessary. When writing complete functions directly in asm rather than using inline asm in C/C++, functions that want to use ZA must call `__arm_tpidr2_save` first. Also, `__arm_tpidr2_save` is a function that is provided by the runtime library, rather than a compiler intrinsic. Because the Arm Compiler 6 does not have the runtime library till now, it is better to use the Clang to compile the function.

3.3.2 Controlling the use of streaming mode

The ABI defines which state `PSTATE.SM` is in on entry to a function and on a normal return, See SME ZA storage. At any time, the processor is either in streaming mode (`PSTATE.SM==1`) or in non-streaming mode (`PSTATE.SM==0`).

1. When writing directly in assembly code, use the instruction `SMSTART SM` to enter streaming mode. Use `SMSTOP SM` to return to non-streaming mode.
2. When writing in C/C++ with ACLE intrinsics, ACLE provides 2 attributes that specify how the program executes statements. The attributes are streaming and streaming-compatible.

- Streaming: shows that the function can only be executed in streaming mode. Entering streaming mode is required, if PE is in non-streaming mode.
- Streaming-compatible: shows that the function can be executed in both streaming and non-streaming modes.


Note

Non-streaming is default, which shows that the function can only be executed in non-streaming mode. An exit from streaming mode is required, if PE is in streaming mode and you want to execute in non-streaming mode.

The following code example shows how to define the execution statements of a function with attributes.

```
<...>
// "n" stands for "non-streaming"
// "s" stands for "streaming"
// "sc" stands for "streaming-compatible"

void n_callee(void); // The program is in non-streaming statements when calling this
 function
```

```
void s_callee(void) __arm_streaming; // The program is in streaming statements when
 calling this function
void sc_callee(void) __arm_streaming_compatible; // The program is in streaming-
compatible statements when calling this function
<...>
```

The code below shows how to use attributes to specify whether code can or must execute in streaming mode. The code uses the SME intrinsic `__arm_in_streaming_mode`, which returns true if the caller is executing in streaming mode. The program automatically switches mode as necessary before calling a function and restores the previous mode on return.

```
<...>
#include <stdlib.h>
#include <stdio.h>
#include <arm_sme.h>

void n_print_sme_status(void){
   printf ("__arm_in_streaming_mode = %d \n", __arm_in_streaming_mode());
}

void s_print_sme_status(void) __arm_streaming{
   printf ("__arm_in_streaming_mode = %d \n", __arm_in_streaming_mode());
}

void sc_print_sme_status(void) __arm_streaming_compatible{
   printf ("__arm_in_streaming_mode = %d \n", __arm_in_streaming_mode());
}

int main(){
  n_print_sme_status();
  s_print_sme_status();
  sc_print_sme_status();
  return 0;
}
<...>
```

The following output is generated:

```
__arm_in_streaming_mode = 0
__arm_in_streaming_mode = 1
__arm_in_streaming_mode = 0
```

The caller `s_print_sme_status`, a function with streaming attribute, calls `__arm_in_streaming_mode`. This returns true because the caller is streaming. While the caller `n_print_sme_status` is without a streaming attribute, which means the caller is non-streaming, the output shows false. Because the `main()` is non-streaming, the `sc_print_sme_status` runs in non-streaming mode.

3.3.2.1 Managing streaming mode across function boundaries

ACLE provides attributes for managing streaming modes across function boundaries, which extend to 3 function types - `non-streaming types`, `streaming types` and `streaming-compatible type`.

The function type classification decides which mode the program is in on entry to the function and which mode the program is in on return from the function.

For example in Controlling the use of streaming mode, if the non-streaming function is called during a streaming function, the mode is streaming when entering into the `s_print_sme_status()` and is then switched into non-streaming when branching into `n_print_sme_status()`. After completing the non-streaming function, the mode returns to streaming.

```
<...>
void s_print_sme_status(void) __arm_streaming{
  n_print_sme_status();
  printf ("__arm_in_streaming_mode = %d \n", __arm_in_streaming_mode());
}

...
// Result of printf
//__arm_in_streaming_mode = 0
//__arm_in_streaming_mode = 1
<...>
```

Calls can be from a streaming caller to a non-streaming callee or from a non-streaming caller to a streaming callee. In both cases, these changes of mode are automatic, and it is the compiler's responsibility to insert the necessary instructions. There are no intrinsics that map directly to `SMSTART` and `SMSTOP`.

In addition to being changed across function boundaries, streaming mode can be changed locally by adding an `__arm_locally_streaming` attribute to a function. The program automatically puts the program into streaming mode before executing the statements and restores the previous mode afterwards. This choice is internal to the function definition. It is not visible to callers and so it can be changed without affecting the function's binary interface. Also, the `__arm_locally_streaming` attribute can override the function type classification. See the specific description of changing streaming mode locally at section 15.1.2 in ACLE.

Streaming mode has some effects on code:

- It can change the length of SVE vectors and predicates. The streaming vector length (SVL) might be different from the normal non-streaming vector length (VL). When switching modes, the undefined behavior needs to be worried except the SVL and VL are equal. For example, in streaming-compatible code, when switching from streaming to non-streaming modes, Z and P register states are not maintained. Therefore, it is not guaranteed to pass the values initialized on Z with VL from non-streaming to streaming with SVL.

  In the following code example, `ptr` is a pointer of non-streaming vector. Function `s_add` adds 1 to all the elements of x, which is a streaming vector. The call from a non-streaming function `ns_add()` to a streaming function `s_add()` is undefined if the SVL and VL are not equal.

  ```
  svint32_t s_add(svint32_t x) __arm_streaming {
    return svadd_x(svptrue_b8(), x, 1);
  }

  void ns_add(svint32_t *ptr) {
  ```

```
  *ptr = s_add(*ptr);
}
```

`svint32_t` defines the signed integer data type for single vector. `svptrue_b8` returns a predicate in which every element is true. `svadd_x` adds 1 to every active element of x.

- SME streaming intrinsics in ACLE can only be executed in streaming mode, while non-streaming intrinsics can only be executed in non-streaming mode.

3.3.3 Controlling the use of ZA storage

ZA storage enablement is controlled by another processor state bit - `PSTATE.ZA`. Like the streaming mode, there are two ways to enable ZA storage. The first one is through `SMSTART` instruction written directly in assembly. The second one is to use the attributes with ACLE intrinsics. In C and C++ code, access to ZA is controlled at the function granularity. AAPCS gives a one-in-three choice for how a function handles ZA.

- The function does not use ZA. This is the default.
- The function uses ZA, which it shares with its caller. This is indicated by the `__arm_shared_za` function type attribute with syntax `__arm_inout("za")`.
- The function uses ZA that it creates `from scratch` and it does not share with its caller. This is indicated by the `__arm_new_za` function type attribute with syntax `__arm_new("za")`.

`__arm_new("za")` is at the beginning when functions is defined. While, `__arm_inout("za")` should be at the end because it describes the type of the function, that callers to the function are aware of. Adding or removing `__arm_inout("za")` changes the interface of the function and the ABI, that would require all callers to be recompiled. However, callees do not need to be recompiled when calling the functions with `__arm_new("za")` attribute. See more explanation at ACLE.

The following code example shows how to enable the ZA storage with `__arm_new("za")` attribute.

```
<...>
#include <stdio.h>

__arm_new("za") void SME_func1(void) __arm_streaming {
    __asm volatile(
    "mov     w12,  #0x0  \n"
    "mov     x13,   #0x80000000  \n"
    "ptrue   pn8.b  \n"
    "ld1b    {z0.b-z3.b}, pn8/z, [x13]  \n"
    "mova    za0h.b[w12, 0:3], {z0.b-z3.b}  \n"
    ::: "x12","x13","p8","z0","z1","z2","z3","za"
    );
}

int main() {
```

```
  SME_func1();
  return 0;
}
<...>
```

In this case, ZA storage is enabled by the attributes `__arm_new("za")`, which instructs the compiler to insert a `SMSTART ZA` instruction at the beginning of the function.

The following example is a complex one to show the effect of ZA storage, when calling functions with different ZA storage attributes.

```
<...>
#include <stdlib.h>
#include <stdio.h>
#include <arm_sme.h>

const int kMaxSVEVecLen = 2048;

static unsigned char zero_buffer[kMaxSVEVecLen] = { 0 };

unsigned long checksum_za(void) __arm_streaming __arm_inout("za") {
  unsigned long i;
  unsigned long svl = svcntsb();
  unsigned char buffer[svl];
  unsigned long sum = 0;

  for (i = 0; i < svl; i++) {
    unsigned long j;
    svstr_vnum_za(i, buffer, 0);       // Store data to buffer with element of the
 i-th row of ZA with offset 0
    for (j = 0; j < svl; j++) {
      sum += buffer[j];
    }
  }

  return sum;
}

void assign_za_1(void) __arm_streaming __arm_inout("za") {
  svbool_t p_all = svptrue_b8();      // Set predicate elements to true
  svint8_t zn = svdup_n_s8(3);        // Set all lanes to the same value that is 3
  svwrite_hor_za8_s8_m(0, 0, p_all, zn);  // Write the vector zn into horizontal ZA
 slices with predication p_all
}

__arm_new("za") void assign_za_2(void) __arm_streaming {
  svbool_t p_all = svptrue_b8();
  svint8_t zn = svdup_n_s8(4);
  svwrite_hor_za8_s8_m(0, 0, p_all, zn);
}

void assign_za_3(void) __arm_streaming __arm_inout("za") {
  svbool_t p_all = svptrue_b8();
  svint8_t zn = svdup_n_s8(4);
  svwrite_hor_za8_s8_m(0, 0, p_all, zn);
}

unsigned long before[3], after[3];

__arm_new("za") void ZA_attribute_test(void) __arm_streaming {
  svzero_za();                  // Clean ZA.

  before[0] = checksum_za();    // Check the sum of a vector in ZA. Since the ZA is
 cleaned, the sum is 0.
  assign_za_1();                // The first assignment of ZA. The value of each
 element is 3 with length of 8 bits.
```

```
  after[0] = checksum_za();     // Check the sum of a vector in ZA. Since SVL is 512
 bit, each element is 3 with length of 8 bits, the sum of a vector is 192 (512/8*3).

  before[1] = after[0];
  assign_za_2();                // Another new ZA is created. The value of each
 element is 4 with length of 8 bits.
  after[1] = checksum_za();     // Checksum remains 192 for a SVL=512 bits since the
 original ZA state is not changed.

  before[2] = after[1];
  assign_za_3();
  after[2] = checksum_za();     // The sum of a vector is 256 (512/8*4).
}

int main() {
  ZA_attribute_test();
  printf("Called assign_za_1. Before = %ld, after = %ld\n", before[0], after[0]);
  printf("Called assign_za_2. Before = %ld, after = %ld\n", before[1], after[1]);
  printf("Called assign_za_3. Before = %ld, after = %ld\n", before[2], after[2]);
  return 0;
}
<...>
```

The following result is obtained:

```
Called assign_za_1. Before = 0, after = 192
Called assign_za_2. Before = 192, after = 192
Called assign_za_3. Before = 192, after = 256
```

In this example, the callees with `__arm_inout("za")` attribute like `check_sum_za()`, `assign_za_1()`, and `assign_za_2()` have the ZA state, which they share with their caller `ZA_attribute_test()`. The result also shows that the functions with `__arm_inout("za")` have changed ZA register and the sum result. The callee with `__arm_new("za")` like `assign_za_2()` creates another new ZA and change its state. This does not affect the original ZA stated created by `ZA_attribute_test()`. Therefore, the sum does not change after executing `assign_za_2()`.

Functions that use ZA can also use the SME instruction intrinsics to manipulate that state. These intrinsics act as shared-ZA functions and share ZA state with their callers, such as `svwrite_hor_za8_s8_m()` which change the ZA state created by `assign_za_2()`'s attribute.

`svzero_za` clears the ZA content. `svldr_vnum_za` does slice_offset filling of horizontal ZA slices with data in a vector. `svstr_vnum_za` does slice_offset filling of a vector with data in horizontal ZA slices. `svdup_n_s8` sets all lanes of a SVE vector with specific value. `svwrite_hor_za8_s8_m` writes the horizontal ZA slices with a SVE vector. For more information about intrinsics, refer to the web page and ACLE.

3.4 How to run an SME application

Without access to SME hardware, you can use models or emulators to develop self-designed application. For all the examples in chapter Toolchains and model support, you can use the Fast Models `FVP_Base_RevC-2xAEMvA`, downloaded from Fixed Virtual Platforms, or the

`Base_AEMvAx1_SVE`, pre-installed at Arm Development Studio. Both options support modeling on Arm AArch64 platforms with SME/SME2.

The following code shows the basic model parameters setting.

```
// Command to execute the executable file on FVP
<...>
$AEM/models/Linux64_GCC-9.3/FVP_Base_RevC-2xAEMvA                    \
--plugin=$AEM/plugins/Linux64_GCC-9.3/ScalableVectorExtension.so     \
-C cluster0.NUM_CORES=1                                              \

-C cluster1.NUM_CORES=0                                              \

-C semihosting-enable=1                                              \

-C cluster0.has_arm_v9-2=1                                           \

-C SVE.ScalableVectorExtension.has_sme=1                             \

-C SVE.ScalableVectorExtension.enable_at_reset=1                     \

-C bp.secure_memory=false                                            \

-a <executable file>
<...>
```

The table shows the meaning of each parameter, when setting the basic FVP model.

Parameter name	Description
plugin	Specify the plugin
cluster0.NUM_CORES	Set number of cores in cluster 0
cluster1.NUM_CORES	Set number of cores in cluster 1
semihosting-enable	Enable semihosting for all cores
cluster0.has_arm_v9-2	Implement the ARMv9.2 Extension. FEAT_SME is implemented at Armv9.2-A
SVE.ScalableVectorExtension.has_sme	Whether SME is implemented (`SVE.ScalableVectorExtension.has_sme2` needs to be 1 if SME2 is implemented)
SVE.ScalableVectorExtension.enable_at_reset	Start with system registers set up for Scalable Vector Extension use
bp.secure_memory	Disable security checking by TZC-400

You can download the application image directly into the fast model, execute, and debug through debugger. See Debug tools. You need to set more parameters. For example, `SVE.ScalableVectorExtension.sme_veclens_implemented` controls the SME vector lengths. For more information, see **Plugins for Fast Models** in Fast Model Reference Guide .

3.5 Debug tools

Debug tools such as Arm Development Studio Debugger can provide visibility of vectors and ZA tiles.

For Arm Development Studio debugger, you can debug the SME/SME2 application code because the Arm Debugger is intended as a bare metal debugger to debug the bootloaders and kernels

including Linux and RTOS. Therefore, you can get information such as if the state is correctly saved, restored on task switch or system calls, display the register contents such as Z vectors and ZA tile elements and print the ZA tile contents as a matrix.

For application debugger GDB or LLDB, you can display the Z vectors and ZA tile elements in the appropriate format. In addition, GDB's python support can print the tile contents as a matrix or export as JSON or CSV format for analysis in another tool.

The figure 3-3 shows how to set the debug configuration, and debug the example through Arm Development Studio Debugger. The figure 3-4 shows to check the value changes of ZA tiles.

Figure 3-3: Chart for debug configuration of SME example

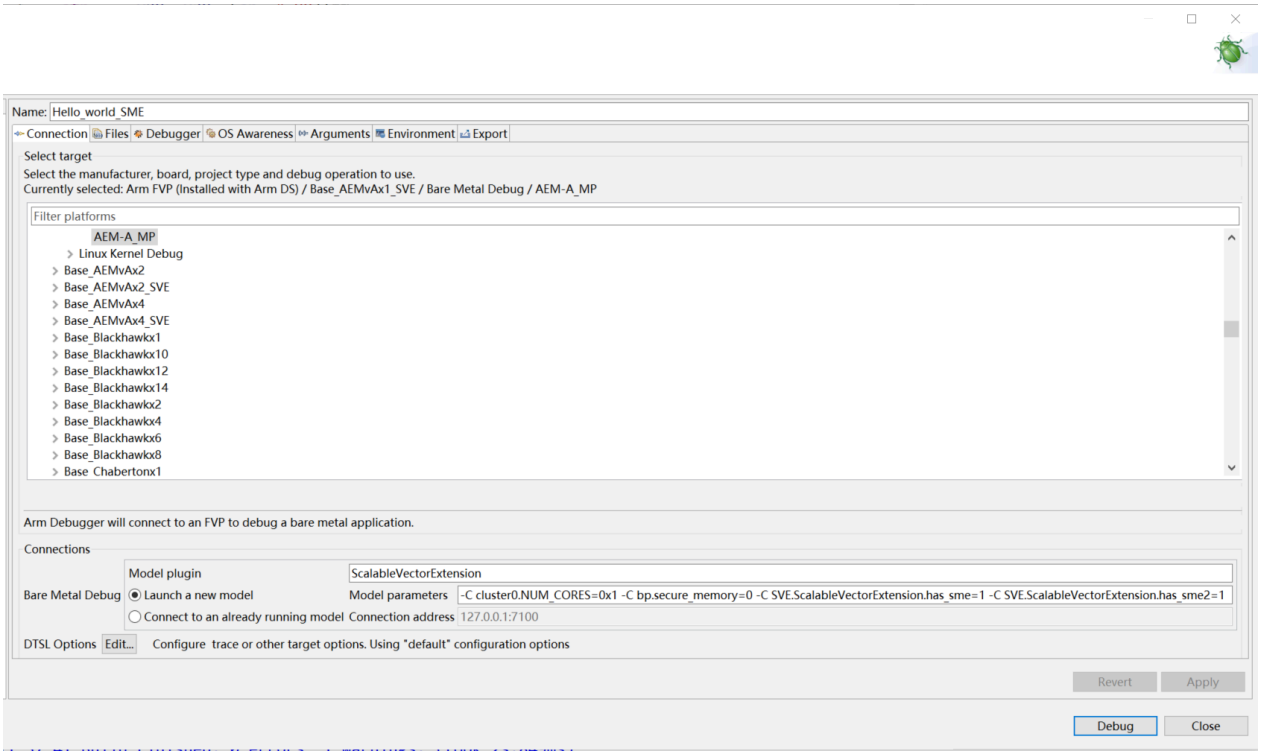
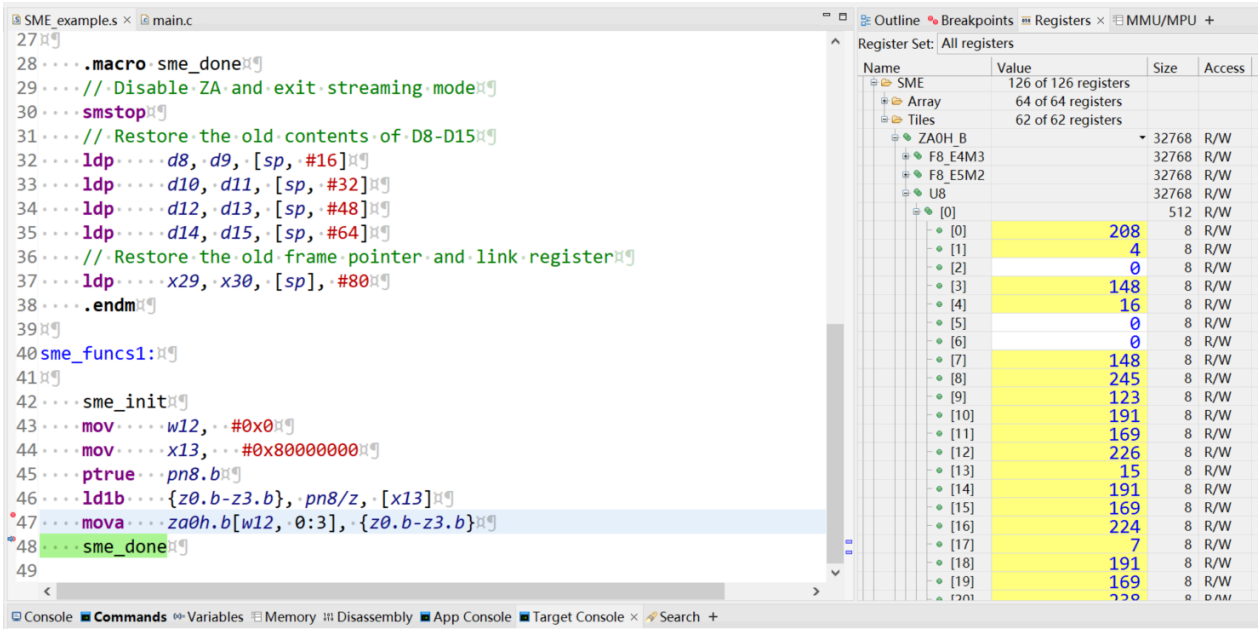


Figure 3-4: Chart for debug information example



4. SME2 code examples

The following sections show examples of assembly code written using SME2 instructions. These examples can help you learn about SME2 so that you can apply the techniques in your own code.

The examples are:

- Matrix-by-matrix multiplication:
  - matmul_fp32: Single precision matrix-by-matrix multiplication.
  - matmul_int8: 8-bit integer to 32-bit integer matrix-by-matrix multiplication.
- Matrix-by-vector multiplication:
  - gemv_cm_int8: 8-bit integer to 32-bit integer matrix-by-vector multiplication.
- Compressed matrix-by-vector multiplication:
  - lut_gemv_rm_int8: Compressed 8-bit integer to 32-bit integer matrix-by-vector multiplication.
- Complex-valued matrix-by-matrix multiplication:
  - cplx_matmul_fp16fp32: Complex-valued half-precision to single precision floating-point matrix-by-matrix multiplication.

5. matmul_fp32: Single precision matrix-by-matrix multiplication

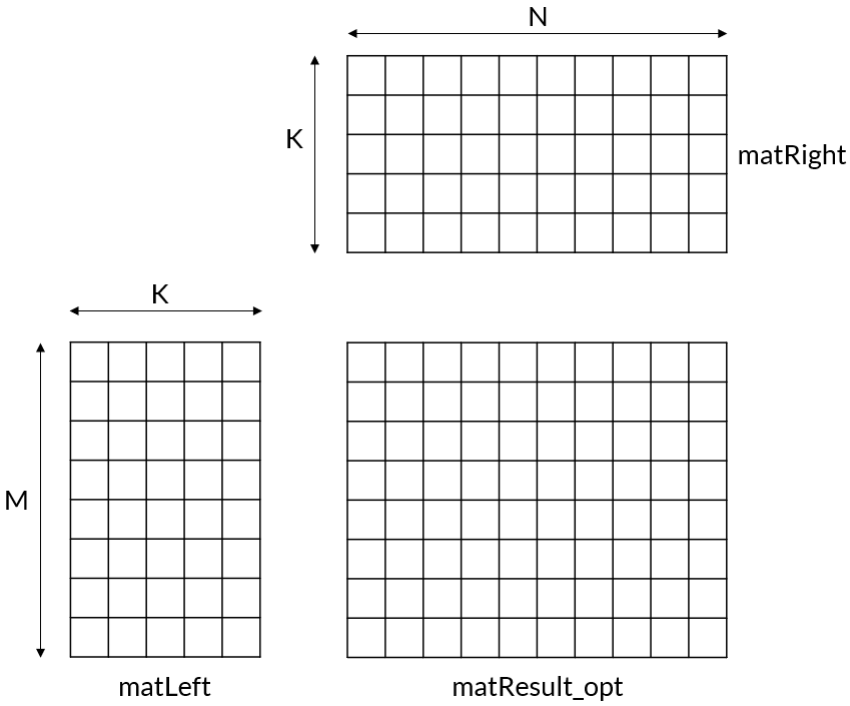
The matmul_fp32 example implements vector length agnostic, single-precision floating-point matrix-by-matrix multiplication using the `fmopa` outer product instruction.

The explanation focuses on the SME2-specific parts of the example: the left-hand side (LHS) matrix preprocessing function `preprocess_l` and the SME2-optimized matrix multiplication function `matmul_opt`.

5.1 Overview of the matmul_fp32 algorithm

The matmul_fp32 example uses the fact that multiplying two matrices together is the same as summing the outer products for each column of `matLeft` and each row of `matRight` in turn. The code multiplies the matrices `matLeft` and `matRight` to produce the result `matResult_opt`:

Figure 5-1: Example matrices



- `matLeft` is an M x K LHS input matrix in row-major format.
- `matRight` is an K x N RHS input matrix in row-major format.
- `matResult_opt` is an M x N matrix containing the result of multiplying `matLeft` with `matRight` in row-major format.

The initial input matrices are stored in memory as row-major arrays. Matrix multiplication is performed as the sum of the outer product of one column from `matLeft` and one row from `matRight`. Because the outer product requires column elements from `matLeft`, the code rearranges the `matLeft` data so that that the column elements are stored contiguously in memory.

The implementation therefore has two steps:

1. Rearrange the LHS `matLeft` matrix, using the function `preprocess_l`.

   The `matLeft` matrix is transpose-like rearranged and stored to memory. More precisely, blocks of SVLs (rows) x K (columns) are transposed by data tiling and contiguously stored to memory. This rearranged matrix is called `matLeft_mod`.
2. Multiply the rearranged `matLeft_mod` matrix and `matRight` matrix using the function `matmul_opt`.

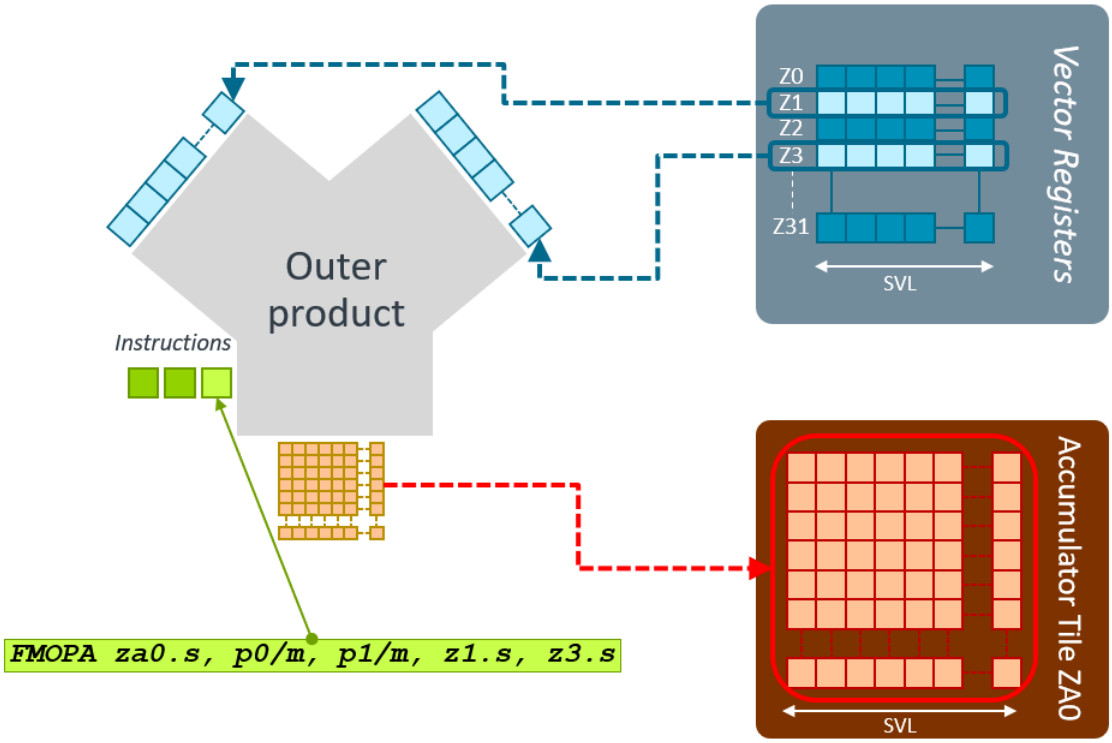
   This function contains three nested loops:

   a. The outermost loops iterates over the rows of the result matrix.
   b. The middle loop iterates over the columns of the result matrix.
   c. The innermost loop iterates over the K dimension, producing result matrix elements as a sum of products.

These operations are examined in more detail in the preprocess_l function details and matmul_opt function details sections.

The outer product calculation uses the `fmopa` instruction. Figure 5-2: Outer product operation using fmopa instructions on page 59 shows each `fmopa` instruction reading two SVE Z input vectors and updating an entire SME ZA tile:

Figure 5-2: Outer product operation using fmopa instructions



5.2 preprocess_l code

The following code shows the `preprocess_l` function with numbered lines. Subsequent sections explain how the code operates.

```
1.  preprocess_l:       // x0: M, x1: K, x2: &matLeft, x3: &matLeft_mod
2.      smstart
3.
4.  // constants
5.      cntw    x4                      // SVLs
6.      mul     x11, x4, x1             // SVLs*K
7.      lsl     x14, x1, #1             // 2*K
8.      add     x15, x14, x1            // 3*K
9.
10.     mul     x16, x4, x4             // SVLs*SVLs
11.
12.     mov     x7, #0
13.     whilelt p0.s, x7, x0            // Tile predicate (M dimension)
14.
15. .Loop_outer:
16.     mov     x8, x2                  // matLeft load base address
17.     mov     x9, x3                  // matLeft_mod store base address
18.     add     x5,  x2, x1, lsl #2     // Exit condition for inner loop
19.
20.     add     x10, x9 , x11, lsl #2   // 32b Tile0 store predicate condition
21.     sub     x13, x10, x16, lsl #2   // 32b Tile1 store predicate condition
22.     whilelt pn8.b, x8, x5, vlx2     // Tile predicate-as-counter (K dimension)
23.
24. .Loop_inner:
```

```
25.     mov     x6, x8                  // matLeft
26.
27.     mov     w12, #0                 // Load_loop counter
28.
29. .Load_loop:
30.     psel    pn10, pn8, p0.s[w12, 0]
31.     psel    pn11, pn8, p0.s[w12, 1]
32.     psel    pn12, pn8, p0.s[w12, 2]
33.     psel    pn13, pn8, p0.s[w12, 3]
34.     ld1w    {z20.s, z28.s}, pn10/z, [x6]                // matLeft
35.     ld1w    {z21.s, z29.s}, pn11/z, [x6, x1,  lsl #2]   // matLeft + K
36.     ld1w    {z22.s, z30.s}, pn12/z, [x6, x14, lsl #2]   // matLeft + K*2
37.     ld1w    {z23.s, z31.s}, pn13/z, [x6, x15, lsl #2]   // matLeft + K*3
38.     mova    za0h.s[w12, 0:3], {z20.s-z23.s}
39.     mova    za1h.s[w12, 0:3], {z28.s-z31.s}
40.
41.     add     x6, x6, x1, lsl #4      // matLeft+=4*K FP32 elements (bytes)
42.     add     w12, w12, #4            // Increment counter
43.     cmp     w12, w4
44.     b.mi    .Load_loop
45.
46.     mov     w12, #0                 // Store_loop counter
47.
48. .Store_loop:
49.     whilelt pn10.b, x9, x10, vlx4
50.     whilelt pn11.b, x9, x13, vlx4
51.     mova    {z0.s-z3.s}, za0v.s[w12, 0:3]
52.     mova    {z4.s-z7.s}, za1v.s[w12, 0:3]
53.     st1w    {z0.s-z3.s}, pn10, [x9] // Store 4 col vectors to matLeft_mod
54.     st1w    {z4.s-z7.s}, pn11, [x9, x16, lsl #2]  // matLeft_mod+SVLs*SVLs
55.     addvl   x9, x9, #4              // matLeft_mod += 4*SVLb (bytes)
56.     add     w12, w12, #4            // Increment counter
57.     cmp     w12, w4
58.     b.mi    .Store_loop
59.
60.     add     x9, x9, x16, lsl #2
61.     addvl   x8, x8, #2              // matLeft+= 2*SVLb (bytes)
62.     whilelt pn8.b, x8, x5, vlx2
63.     b.first .Loop_inner
64.
65.     add     x3, x3, x11, lsl #2     // matLeft_mod+= SVLs*K FP32 elms (bytes)
66.     add     x2, x2, x11, lsl #2     // matLeft+= SVLs*K FP32 elms (bytes]
67.     incw    x7
68.
69.     whilelt p0.s, x7, x0
70.     b.first .Loop_outer
71.
72.     smstop
73.
74.     ret
```

5.3 preprocess_l function overview

The `preprocess_l` function rearranges the `matLeft` matrix so that blocks of SVLs (rows) x K (columns) are transposed by data tiling and contiguously stored to memory. This rearrangement is implemented by loading the `matLeft` matrix rows to horizontal slices of a 32-bit ZA tile and storing vertical slices of that 32-bit ZA tile contiguously to memory. The input matrix is zero-padded to a multiple of SVLs rows.

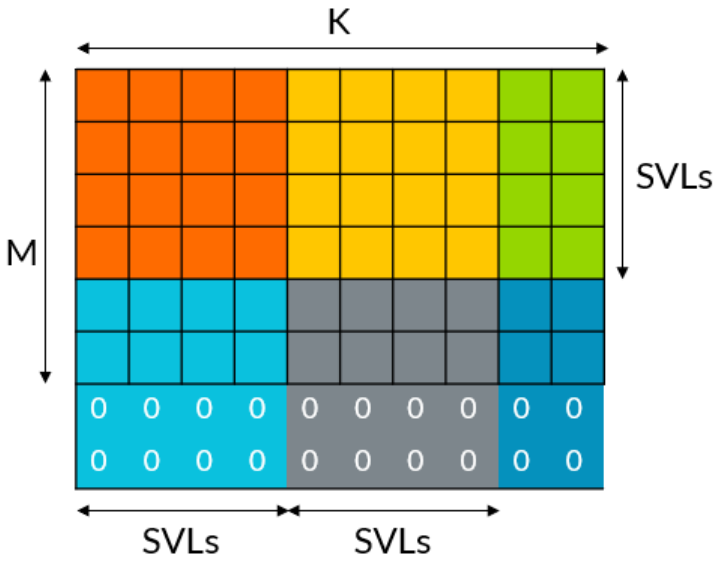
Rearranging the matrix data in memory in this way makes the subsequent memory accesses in the matrix multiplication calculation more efficient, because all data is then read from contiguous memory addresses.

Note

SVLs is the number of 32-bit elements in the Streaming SVE vector length, which is **IMPLEMENTATION DEFINED**. By basing our calculations on this value rather than a hard-coded value, the code is vector length agnostic to maximize efficiency on any platform, regardless of the implemented vector length. That is, the code makes the most efficient use of the vector registers regardless of their implemented size.

Figure 5-3: Matrix divided into blocks on page 61 shows how the entire M x K `matLeft` input matrix is divided into SVLs x SVLs blocks, with elements outside of the matrix zeroed out.

Figure 5-3: Matrix divided into blocks



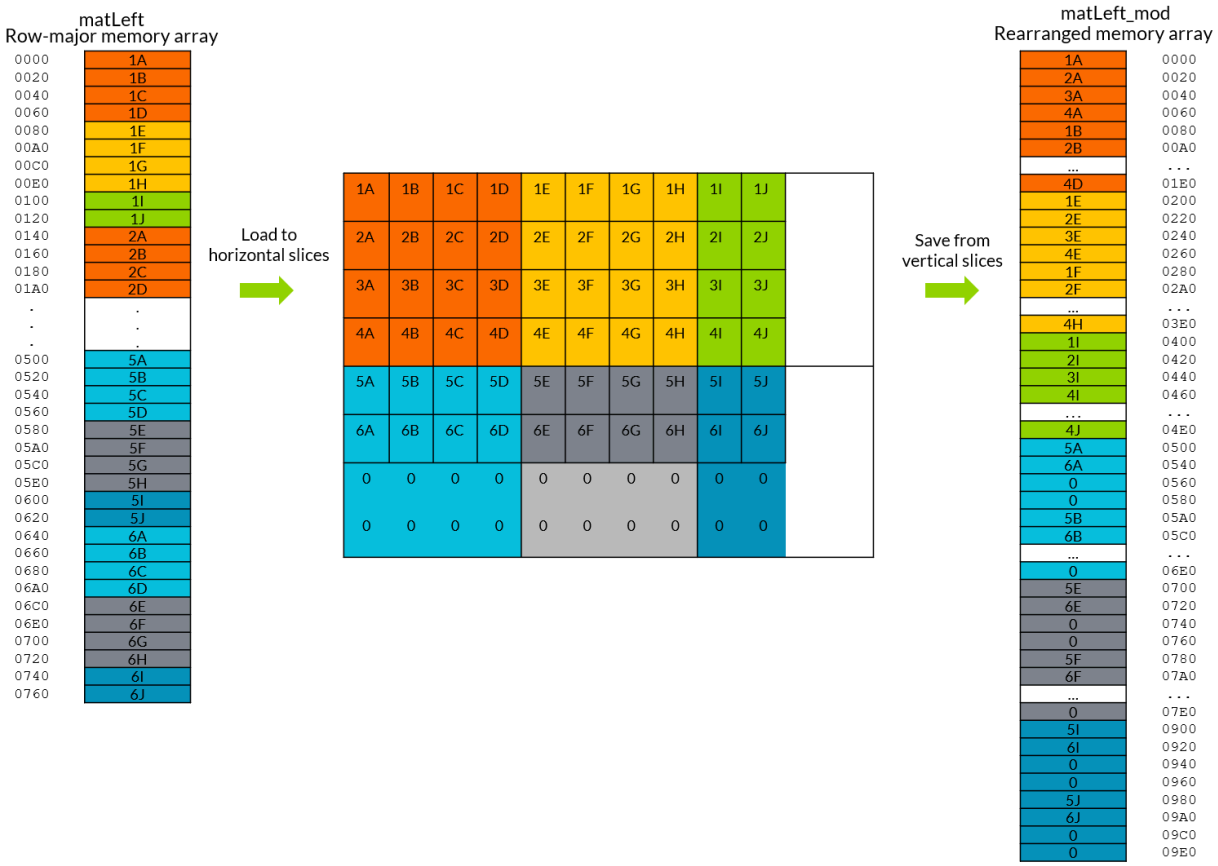
The following shows the overall structure of the `preprocess_l` function:

```
Loop_outer:
  // Iterate over the rows of the input matrix, the M dimension.
  // Each iteration of Loop_outer rearranges SVLs rows of the input
  // matrix.

  Loop_inner:
    // Iterate over the columns of the input matrix, the K dimension.
    // Each iteration of Loop_inner deals with a group of 2*SVLs
    // columns from the original matrix.

    Load_loop:
      // Loads a segment of SVLs (rows) x 2*SVLs (columns) from
      // the input matrix to two 32-bit ZA tiles.
      //
      // Each iteration of Load_loop loads 2*SVLs elements from
      // 4 rows horizontally into two 32-bit ZA tiles.
      //
      // In Load_loop, input matrix loads are predicated using
      // predicate-as-counter. This zeroes inactive elements in
      // the destination vector.

    Store_loop:
      // Stores vertical slices of the two 32-bit element ZA tiles to
      // memory.
```

```
        //
        // Each iteration of Store_loop takes 4 vertical tile slices from
        // the 32-bit ZA tiles and stores the elements at consecutive
        // memory locations.
        //
        // In Store_loop, output matrix stores are predicated using
        // predicate-as-counter. If the input matrix rows were
        // zero-padded to a multiple of SVLs rows, those padded zeros
        // are also stored.
```

By repeatedly writing to horizontal slices of the ZA tile on load, but reading from vertical slices to store, the matrix is rearranged in blocks as shown in Figure 5-4: Rearranging the matLeft matrix on page 62:

Figure 5-4: Rearranging the matLeft matrix



The left-hand side of the figure shows the row-major input matrix in consecutive memory address locations, with each element occupying 4 bytes. The right-hand side of the figure shows the rearranged data, after processing by the `preprocess_l` function. Rearranging the data in this way means that the `matmul_opt` function accesses `matleft_mod` data from contiguous memory addresses when performing the matrix multiplication, improving the efficiency of the algorithm.

The example in this figure assumes that SVL is 128b, with 4x4 32-bit elements per ZA tile and that the dimensions of the `matLeft` matrix are 6 rows by 10 columns.

The preprocess_l function details section describes specific portions of the code in more detail.

5.4 preprocess_l function details

This section describes how the `preprocess_l` function operates, looking at sections of the code in turn.

- On entry, function arguments are passed in registers as follows:
  - x0: M, the number of rows in the `matLeft` matrix
  - x1: K, the number of columns in the `matLeft` matrix
  - x2: The base address of the input matrix, `matLeft`
  - x3: The base address of the output matrix, `matLeft_mod`
- Line 2:

  Enters Streaming SVE mode and enables the ZA array storage.

  The `smstart` instruction is only required if you are not already in streaming mode. This example assumes that streaming SVE mode is locally managed using `smstart` and `smstop` instructions. This example requires streaming SVE mode to enable access to ZA array storage.

- Lines 12-13:

  Register x7 is used as the outer loop counter.

  Predicate register p0 determines when the bounds of the input array `matLeft` are exceeded, and therefore when to zero inactive elements in `Load_loop`.
- Lines 15-27 and 60-70 (`Loop_outer` and `Loop_inner`)

  `.Loop_outer` iterates through the rows of the input matrix, the M dimension. Each iteration of `.Loop_outer` rearranges SVLs rows of the input matrix and calculates the start and end address in memory of a row to prepare to divide sub-sections processed by `.Loop_inner`. The end address, x5, is calculated by taking the start address and adding the number of columns (K) from x1 multiplied by 4, because each element contains 32 bits, or 4 bytes.

  `.Loop_inner` iterates over column elements in groups of 2 x SVLs elements. That is, for a matrix with K columns, if K is greater than 2 x SVLs, each row is processed using multiple iterations of `.Loop_inner`.

  Each iteration of `.Loop_inner` uses a predicate-as-counter register, pn8, to identify which columns should be processed. If the number of columns, K, is not an exact multiple of 2 x SVLs, then the final iteration of `.Loop_inner` uses partially full predicate registers to identify the leftover columns. All other iterations use full predicate registers to process all elements. The `whilelt pn8.b, x8, x5, vlx2` instruction generates the predicate-as-counter, which identifies the number of elements processed by the 2-vector load instructions in each Loop_inner

iteration. You can also think of x8 as a `.Loop_inner` loop counter in bytes. While this counter is less than the end of the row address, then predicate elements are marked as true. Each iteration processes a maximum of 2 x SVLb bytes. The loop counter is incremented by SVLs at the end of each iteration in line 61.

`Loop_outer` uses a predicate-as-mask, or single vector predicate, to identify which rows to process. The predicate-as-mask is set by the `whilelt p0.s, x7, x0` instruction, where x7 is the current `Loop_outer` counter and x0 is the number of rows, M. The loop counter is incremented by SVLs at the end of each iteration in line 67, `incw x7`.

At the end of each iteration of `Loop_inner`, the base address is incremented by the maximum number of bytes consumed in each iteration, which is 2 x SVLb. When the base address becomes greater than the end address, the predicate becomes all false, and the loop ends.

The exit condition for `Loop_inner` is when all columns have been processed, and the exit condition for `Loop_outer` is when all rows have been processed.

- Lines 29-44 (`Load_loop`)

  Each iteration of `Load_loop` loads 2 x SVLs elements from four rows to consecutive horizontal slices of a 32-bit ZA tile.

  Vertical and horizontal matrix edges are carefully managed by the `psel` instructions in lines 30 to 33. The `psel` instructions in lines 30-33 set four predicate registers based on the value of the four predicate elements in p0 starting at index w12. If the element at index w12 is active, the predicate pn10 is set to the value of pn8. Otherwise, the predicate is set to all false. The same process applies for the other three predicate registers, pn11, pn12, and pn13. This is referred to as 2D predication. This allows unrolling of inner loops while keeping the processing generalized and the code vector length agnostic. In this case, four consecutive rows are processed in each iteration, as long as four rows are available for processing. When the number of remaining rows is less than four, `psel` generates an all false predicate for the un-needed rows, which prevents the load.

  The `ld1w` instructions at lines 34 to 37 perform two-vector loads from four consecutive rows of the input matrix data to a strided pair of vectors, predicated using the predicate-as-counter registers set by the `psel` instructions above. Then the `mova` instructions at lines 38 and 39 move a consecutively numbered group of four vectors to horizontal slices of the 32-bit tiles ZA0 and ZA1. The first half of each loaded segment is moved to ZA0, and second half of the segment is moved to ZA1, achieved by loading to strided vectors and moving consecutively numbered vectors.

  In `Store_loop` we will see that the store instructions use the `v` suffix to access the data in the ZA tiles as vertical columns and rearrange the matrix.

  Each iteration of `Load_loop` then increments the base address in x6 by 4 entire rows, and the ZA tile horizontal slice counter in w12 by 4.

- Lines 46-58 (`Store_loop`)

  `Store_loop` stores two ZA tiles (ZA0 ad ZA1) to memory. Each iteration stores 4 vertical tile slices from each ZA tile into consecutive memory locations, using predication to determine when the bounds of the array are exceeded.

The `whilelt` instructions in lines 49-50 generate predicate-as-counters for four-vector consecutive stores. The `vlx4` in these instructions is the vector length specifier, and indicates that the predicate can control four vectors. The predicates ensure that non-existent columns in the ZA tile are not stored to memory, using the base address to calculate the store bounds for each of the ZA tiles.

The `mova` instructions in lines 51-52 move four vertical slices of ZA0 and ZA1 to consecutive group of Z vectors. The `v` suffix accesses the data in the ZA tile as columns, rearranging the matrix.

The `st1w` instructions in lines 53-53 perform four vector contiguous stores from the Z vectors to memory.

5.5 matmul_opt code

The following code shows the function `matmul_opt` with numbered lines. Subsequent sections explain how the code operates.

```
1.   matmul_opt: // x0: M, x1: K, x2: N, x3: matLeft_mod, x4: matRight, x5:
 matResult
2.      stp     x19, x20, [sp, #-48]!
3.      stp     x21, x22, [sp, #16]
4.      stp     x23, x24, [sp, #32]
5.
6.      smstart
7.
8.      // constants
9.      cntw    x6                      // SVLs
10.     mul     x22, x6, x1             // SVLs*K
11.     mul     x23, x6, x2             // SVLs*N
12.     add     x18, x23, x2            // SVLs*N + N
13.     add     x11, x4, x2, lsl #2     // Exit condition for N loop
14.     mov     x12, #0
15.     cntb    x6                      // SVLb
16.     mov     x14, #0
17.     ptrue   pn10.b                  // Predicate for SME2 VLx2 (a_ptr loads)
18.     whilelt pn8.s, x12, x0, vlx2    // tiles predicate (M dimension)
19.     sub     w6, w6, #8              // SVLb-8
20.
21.  .Loop_M:
22.     // Extract tile 0/1 and tile 2/3 predicates (M) from vlx2 predicate.
23.     pext    { p2.s, p3.s }, pn8[0]
24.     mov     x16, x4                 // b_base
25.     mov     x9, x5                  // c_base
26.     whilelt pn9.b, x16, x11, vlx2   // tiles predicate (N dimension)
27.
28.  .Loop_N:
29.     mov     x7, x3                  // a_ptr = a_base
30.     mov     x17, x16                // b_ptr = b_base
31.     mov     x10, x9                 // c_ptr0 = c_base
32.
33.     // Extract tile 0/2 and tile 1/3 predicates (N) from vlx2 predicate.
34.     pext    { p0.b, p1.b }, pn9[0]
35.
36.     add     x8, x3, x22, lsl #2     // a_base + SVLs*K FP32 elms (bytes)
37.     addvl   x15, x8, #-1            // Exit condition for K loop
38.     ld1w    {z1.s}, p2/z, [x7]      // Load 1st vector from a_ptr
39.
40.     zero    {za}
```

```
41.    ld1w    {z2.s-z3.s}, pn9/z, [x17]  // Load 2 vectors from b_ptr
42.
43.    fmopa   za0.s, p2/m, p0/m, z1.s, z2.s // ZA0+=1st a_ptr vec OP 1st b_ptr vec
44.    ld1w    {z5.s}, p3/z, [x7, x22, lsl #2]  // Load 2nd vector from a_ptr
45.    addvl   x7, x7, #1                       // a_ptr += SVLb (bytes)
46.
47.  .Loop_K:
48.    fmopa   za2.s, p3/m, p0/m, z5.s, z2.s // ZA2+=2nd a_ptr vec OP 1st b_ptr vec
49.
50.    fmopa   za1.s, p2/m, p1/m, z1.s, z3.s // ZA1+=1st a_ptr vec OP 2nd b_ptr vec
51.    ld1w    {z0.s-z1.s}, pn10/z, [x7]     // Load next 2 vectors from a_ptr
52.
53.    fmopa   za3.s, p3/m, p1/m, z5.s, z3.s // ZA3+=2nd a_ptr vec OP 2nd b_ptr vec
54.    ld1w    {z6.s-z7.s}, pn9/z, [x17, x2, lsl #2] // Load next 2 vecs from b_ptr
55.
56.    fmopa   za0.s, p2/m, p0/m, z0.s, z6.s // ZA0+=1st a_ptr vec OP 1st b_ptr vec
57.    psel    pn11, pn10, p3.s[w14, 0]      // Select predicate-as-counter
58.    ld1w    {z4.s-z5.s}, pn11/z, [x7, x22, lsl #2] // Load next 2 vecs from a_ptr
59.
60.    fmopa   za2.s, p3/m, p0/m, z4.s, z6.s // ZA2+=2nd a_ptr vec OP 1st b_ptr vec
61.    add     x17, x17, x2, lsl #3          // b_ptr += 2*N FP32 elms (bytes)
62.
63.    fmopa   za1.s, p2/m, p1/m, z0.s, z7.s // ZA1+=1st a_ptr vec OP 2nd b_ptr vec
64.
65.    fmopa   za3.s, p3/m, p1/m, z4.s, z7.s // ZA3+=2nd a_ptr vec OP 2nd b_ptr vec
66.    ld1w    {z2.s-z3.s}, pn9/z, [x17]     // Load next 2 vectors from b_ptr
67.
68.    fmopa   za0.s, p2/m, p0/m, z1.s, z2.s // ZA0+=1st a_ptr vec OP 1st b_ptr vec
69.    addvl   x7, x7, #2                    // a_ptr += 2*SVLb (bytes)
70.
71.    cmp     x7, x15
72.    b.mi    .Loop_K
73.
74.    fmopa   za2.s, p3/m, p0/m, z5.s, z2.s // ZA2+=2nd a_ptr vec OP 1st b_ptr vec
75.
76.    fmopa   za1.s, p2/m, p1/m, z1.s, z3.s // ZA1+=1st a_ptr vec OP 2nd b_ptr vec
77.
78.    fmopa   za3.s, p3/m, p1/m, z5.s, z3.s // ZA3+=2nd a_ptr vec OP 2nd b_ptr vec
79.    add     x17, x17, x2, lsl #2          // b_ptr += 2*N FP32 elms (bytes)
80.
81.    cmp     x7, x8
82.    b.pl    .Ktail_end
83.
84.  .Ktail_start:
85.    ld1w    {z1.s}, p2/z, [x7]
86.    ld1w    {z2.s-z3.s}, pn9/z, [x17]
87.
88.    fmopa   za0.s, p2/m, p0/m, z1.s, z2.s
89.    ld1w    {z5.s}, p3/z, [x7, x22, lsl #2]
90.
91.    fmopa   za2.s, p3/m, p0/m, z5.s, z2.s
92.
93.    fmopa   za1.s, p2/m, p1/m, z1.s, z3.s
94.
95.    fmopa   za3.s, p3/m, p1/m, z5.s, z3.s
96.
97.  .Ktail_end:
98.    mov     w13, #0
99.    psel    pn11, pn9, p2.b[w13, 0]
100.   psel    pn12, pn9, p3.b[w13, 0]
101.   // ZA tiles to vecs: z0 = za0h[1], z1 = za1h[1], z2 = za2h[1], z3 = za3h[1]
102.   mova    { z0.b-z3.b }, za0h.b[w13, 0:3]
103.   st1w    { z0.s-z1.s }, pn11, [x10]             // Store to c_ptr0
104.   st1w    { z2.s-z3.s }, pn12, [x10, x23, lsl #2] // Store to c_ptr0+(SVLs*N)
105. .Loop_store_ZA:
106.   psel    pn11, pn9, p2.b[w13, 4]
107.   psel    pn12, pn9, p3.b[w13, 4]
108.   mova    { z0.b-z3.b }, za0h.b[w13, 4:7]
109.   st1w    { z0.s-z1.s }, pn11, [x10, x2, lsl #2]  // Store to c_ptr0+N
110.   st1w    { z2.s-z3.s }, pn12, [x10, x18, lsl #2] // Store to c_ptr0+(SVLs+1)*N
111.
```

```
112.    add     x10, x10, x2, lsl #3    // c_ptr0 += 2*N FP32 elms (bytes)
113.    add     w13, w13, #8
114.
115.    psel    pn11, pn9, p2.b[w13, 0]
116.    psel    pn12, pn9, p3.b[w13, 0]
117.    mova    { z0.b-z3.b }, za0h.b[w13, 0:3]
118.    st1w    { z0.s-z1.s }, pn11, [x10]              // Store to c_ptr0
119.    st1w    { z2.s-z3.s }, pn12, [x10, x23, lsl #2]  // Store to c_ptr0+SVLs*N
120.    cmp     w13, w6
121.    b.mi    .Loop_store_ZA
122.
123.    psel    pn11, pn9, p2.b[w13, 4]
124.    psel    pn12, pn9, p3.b[w13, 4]
125.    mova    { z0.b-z3.b }, za0h.b[w13, 4:7]
126.    st1w    { z0.s-z1.s }, pn11, [x10, x2, lsl #2]  // Store to c_ptr0+N
127.    st1w    { z2.s-z3.s }, pn12, [x10, x18, lsl #2] // Store to c_ptr0+(SVLs+1)*N
128.
129.    addvl   x9, x9, #2
130.    addvl   x16, x16, #2            // b_base += 2*SVLb (bytes)
131.    whilelt pn9.b, x16, x11, vlx2   // tile predicate (N dimension)
132.    b.first .Loop_N
133.
134.    add     x3, x3, x22, lsl #3     // a_base += 2*SVLs*K FP32 elms (bytes)
135.    add     x5, x5, x23, lsl #3     // c_base += 2*SVLs*N FP32 elms (bytes)
136.    incw    x12, all, mul #2        // M loop counter += 2* SVLs
137.    whilelt pn8.s, x12, x0, vlx2    // tiles predicate (M dimension)
138.    b.first .Loop_M
139.
140.    smstop
141.
142.    ldp     x23, x24, [sp, #32]
143.    ldp     x21, x22, [sp, #16]
144.    ldp     x19, x20, [sp], #48
145.
146.    ret
```

5.6 matmul_opt function overview

The `matmul_opt` function does the following:

1. Iterates over the columns of the `matLeft` matrix (`matLeft_mod` buffer) and the rows of the `matRight` matrix (`matRight` buffer)
2. Calculates the outer products
3. Stores the result in `matResult_opt`.

The code in this example uses the fact that multiplying two matrices together is the same as summing the outer products for each row and column in turn. That is, given a matrix `matLeft` with dimensions M x K and a matrix `matRight` with dimensions K x N, the result of multiplying `matLeft` and `matRight` produces a matrix `matResult_opt` with dimensions M x N. The result is calculated as follows:

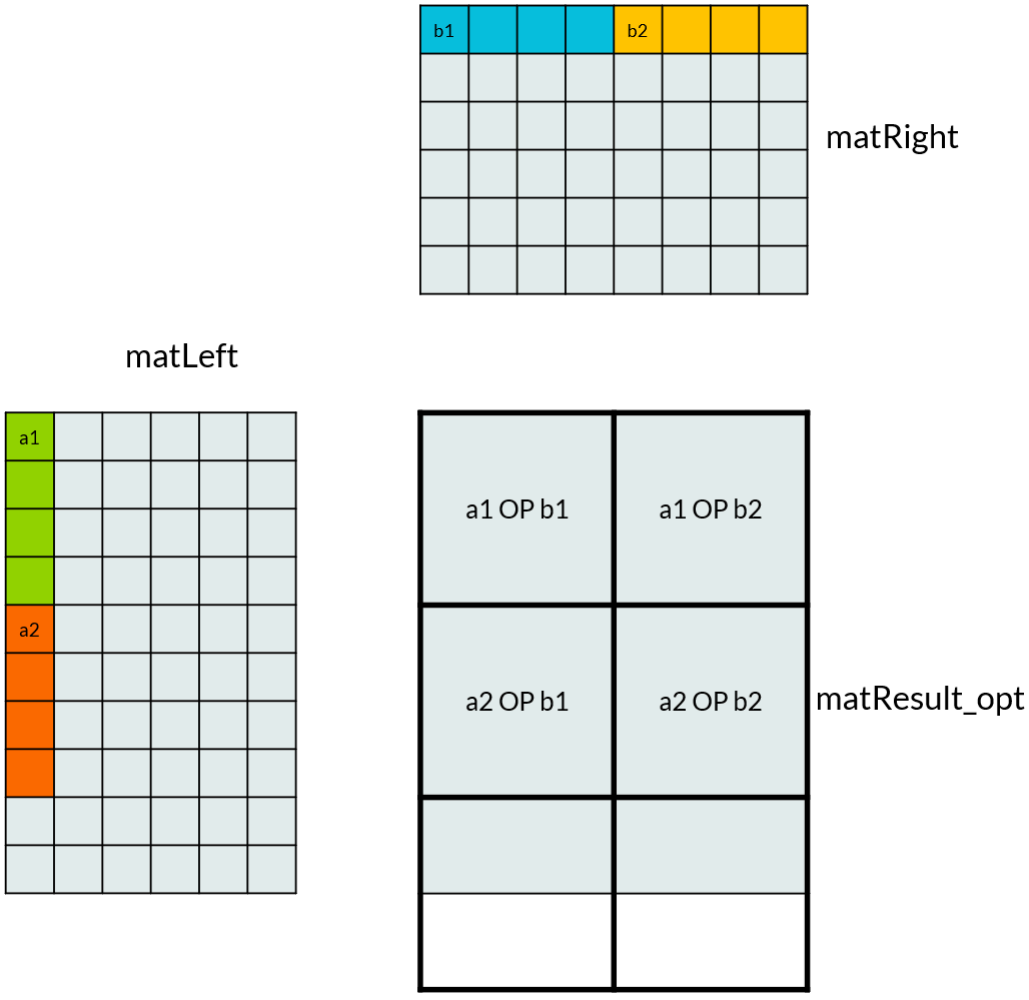
```
for k = 1 to K:
  // Partial products computation
  for m = 1 to M: // m+=2
    for n = 1 to N: // n+=2
      // Inner loops unrolled by 2
      // A OP B is equal to
      matResult_opt(m ,n )   += matLeft(m,k)   x matRight(k,n)
      matResult_opt(m+1,n )  += matLeft(m+1,k) x matRight(k,n)
```

```
        matResult_opt(m  ,n+1)   += matLeft(m,k)    x matRight(k,n+1)
        matResult_opt(m+1,n+1)   += matLeft(m+1,k)  x matRight(k,n+1)
```

`OP` represents the outer product operation.

The `matmul_opt` function uses SME2 functionality that calculates the outer product of two vectors using a single instruction, storing the results in two-dimensional ZA matrix tiles as shown in Figure 5-5: Calculating outer product, 4 tiles at a time on page 68.

Figure 5-5: Calculating outer product, 4 tiles at a time



5.7 matmul_opt function details

This section describes how the `matmul_opt` function operates, looking at sections of the code in turn.

- Lines 2-4:

The code starts by saving registers x19 through x24 to the stack. These registers are restored at the end of the `matmul_opt` function. The Procedure Call Standard for the Arm 64-bit Architecture defines registers x19 through x28 as callee-saved registers, so the `matmul_opt` function must preserve the values of the registers it uses in this range.

- Line 6:

  Enters Streaming SVE mode and enables the ZA array storage.

- Line 18:

  Sets a 32-bit element predicate-as-counter pn8. If the numerical value of x12 is less than the value of base address of `matLeft_mod` with offset 2 * SVLs (because of vlx2), set the active counter of pn8 as the value of x12, otherwise set as vlx2.

- Lines 21-26 and 134-138 (`Loop_M`)

  `Loop_M` iterates over the M dimension in blocks of 2 x SVLs, that is rows in the original `matLeft` matrix.

  The `pext` instruction in line 23 creates predicate-as-mask equivalent predicates, p2 and p3, from the predicate-as-counter register pn8. p2 corresponds to the first SVL, and p3 to the second SVL. These predicate registers control which columns in `matLeft_mod` are processed in each iteration.

  At the end of `Loop_M`, lines 134-135 increment the pointers to the current rows in `matLeft_mod` and `matResult` by the length of the processed data.

  Successive iterations of `Loop_M` update the predicate-as-counter pn8 until all M rows have been processed.


Note

Because the `matLeft_mod` is rearranged from `matLeft`, the columns of `matLeft_mod` are extracted from the rows of `matLeft`.

- Lines 28-45 and 129-132 (`Loop_N`)

  `Loop_N` iterates over the N dimension by a block of 2 * SVLs column. N is the number of columns in the `matRight` matrix.

  The `pext` instruction sets a pair of predicate registers, p0 for the first SVLs columns and p1 for the second SVLs columns, from the predicate-as-counter register pn9. These predicate registers control which columns in `matRight` are processed in each iteration.

  Each iteration of `Loop_N` clears the ZA array by zeroing all elements. The `zero {za}` instruction in line 40 does this, with `za` indicating that the whole ZA array is zeroed.

  At the end of `Loop_N`, lines 129-130 increment the pointers to the current columns in `matRight` and `matResult` by the length of the processed data.

Successive iterations of `Loop_N` update the predicate-as-counter pn9 until all N columns in `matRight` have been processed.

- Lines 47-104 (`Loop_K`)

  `Loop_K` iterates over the K dimension, that is rows in the `matRight` and `matLeft_mod` matrices, computing a sub-block of the result matrix, measuring (2 * SVLs) x (2 * SVLs). These dimensions fit the four 32-bit ZA tiles used to store the results.

  The code uses loop unrolling to improve efficiency. Each `Loop_K` iteration processes two k values, where k=1..K, so that two products are accumulated to each result for each loop iteration.

  The `ld1w` instructions load matrix data from memory to Z vector registers as follows:

  - Z1 and Z0 contain the first SVLs elements from the M dimension of `matLeft` for k and k+1, Z0.s = matLeft[k, 0:SVLs-1] and Z1.s = matLeft[k+1, 0:SVLs-1].
  - Z5 and Z4 contain the second SVLs elements from the M dimension of `matLeft` for k and k +1, Z4.s = matLeft[SVLs:2*SVLs-1, k] and Z5.s = matLeft[SVLs:2*SVLs-1, k+1].
  - Z2 and Z3 contain the 2xSVLs elements from the N dimension of `matRight` for k, Z2.s = matRight[k, 0:SVLs-1] and Z3.s = matRight[k, SVLs:2*SVLs-1].
  - Z6 and Z7 contain the 2xSVLs elements from the N dimension of `matRight` for k+1, Z6.s = matRight[k+1, 0:SVLs-1] and Z7.s = matRight[k+1, SVLs:2*SVLs-1].

  These `ld1w` instructions use predicated 2-vector loads to load two Z registers at a time.

  The single-precision floating-point `fmopa` instructions operate on the 32-bit ZA0, ZA1, ZA2, and ZA3 tiles to compute the outer product of the left and right matrix subblocks as follows:

  - ZA0 contains the outer product: 1st SVLs from M (OP) 1st SVLs from N
  - ZA1 contains the outer product: 1st SVLs from M (OP) 2nd SVLs from N
  - ZA2 contains the outer product: 2nd SVLs from M (OP) 1st SVLs from N
  - ZA3 contains the outer product: 2nd SVLs from M (OP) 2nd SVLs from N

  Each of these `fmopa` instructions are independently predicated, enabling 2D predication of tile results.

  Loads are reused, so that 8 `fmopa` instructions consume 8 loaded vectors (2 for each of left and right matrix per one k), and the load-to-multiply ratio is perfectly balanced in the loop.

  Accumulating these successive outer products takes advantage of the fact that multiplying two matrices together is the same as summing the outer products for each row and column in turn.

  In line 81, note that the function does not use a dedicated loop counter. The code uses the value of left matrix pointer, x7, as both the left matrix load address, and also to determine the `Loop_k` exit condition.

- Line 57

The `psel pn11, pn10, p3.s[w14, 0]` instruction sets the predicate-as-counter pn11 to either all false or all true, depending on the value of the first element of p3. If the first element of p3 is false, then the second SVLs of the `matLeft` column are all zero, so there is no need to load the data. This situation can occur in the final iteration if the number of rows is not exactly divisible by 2 x SVLs.

- Lines 105-127 (`Loop_store_ZA`)

  Finally, all that is required is to store the results to the `matResult` array for each ZA tile.

  The data in the ZA tiles is first transferred from the four tiles to four Z vector registers, using the 8-bit indexed `mova` instruction. Results are stored to `matResult` by updating elements in a segment of memory with size ((2 * SVLs) x (2 * SVLs)). Each row in this segment contains 2 * SVLs elements, formed by combining consecutive slices from 2 tiles:

  - The first half of the memory segment contains rows formed by combining slices from the ZA0 and ZA1 tiles.
  - The second half of the memory segment contains rows formed by combining slices from the ZA2 and ZA3 tiles.

  Consecutive vectors are stored to memory using the 2 vector store instruction `st1w`, predicated using the vlx2 predicate-as-counter.

  The `mova` instruction at line 108 (and elsewhere, due to loop unrolling) illustrates an SME2 coding optimization. The results were obtained in a ZA array using an outer product instruction to 32-bit ZA tiles. However, the `mova` instructions operate on 8-bit tiles (ZA0.B). When 4 consecutive horizontal slices of an 8-bit tile ZA0.B are moved, these slices correspond to one horizontal slice from 4 different 32-bit tiles ZA0.S, ZA1.S, ZA2.S, ZA3.S.

  The `psel` instructions at lines 106-107 (and elsewhere, due to loop unrolling) enable predication of the result matrix stores. This means that the `Loop_store_ZA` loop can cope with situations where the number of rows is not an exact multiple of SVLs by ignoring the leftover rows.

  The `Loop_store_ZA` function iterates over SVLb horizontal row slices from the ZA0.b tile, with each iteration extracting four row slices corresponding to the four ZAx.S 32b tiles.

- Line 140

  Exit Streaming SVE mode, and disable the ZA storage.

- Lines 142-146

  Restore the callee-saved registers and return from the `matmul_opt` function.

6. matmul_int8: 8-bit integer to 32-bit integer matrix-by-matrix multiplication

The matmul_int8 example implements vector length agnostic, unsigned 8-bit integer inputs matrix-by-matrix multiplication. It returns a result matrix of unsigned 32-bit integers.

This guide focuses on the SME2-specific parts of the example:

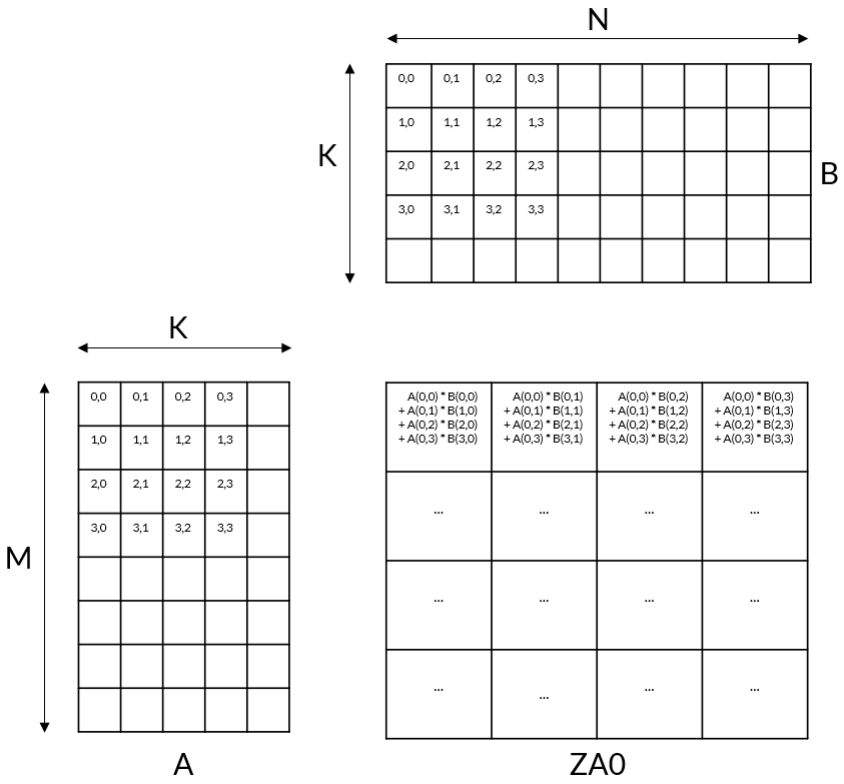
- The `matLeft` matrix preprocessing function `preprocess_l`
- The `matRight` preprocessing function `preprocess_r`
- The SME2-optimized matrix multiplication function `matmul_opt`

6.1 Overview of the matmul_int8 algorithm

The `matmul_int8` example uses the fact that multiplying two matrices together is the same as summing the outer products for each column of `matLeft` and each row of `matRight` in turn, just like the matmul_fp32 example. The code uses the four-way sum of outer products and accumulate `umopa` instruction to multiply the 8-bit `matLeft` and `matRight` matrices and produce the 32-bit result matrix `matResult_opt`.

Figure 6-1: The umopa instruction on page 73 shows how the `umopa` instruction accumulates four 8-bit outer-products into a single 32-bit container:

Figure 6-1: The umopa instruction



For example, consider a matrix multiplication C = A x B, with M = 7, K = 6, and N = 5. The first `umopa` instruction sums four consecutive multiplications over the K dimension in a single instruction, as follows:

```
ZA0.s[0][0] += A(0,0)*B(0,0)  +  A(0,1)*B(1,0)  +  A(0,2)*B(2,0)  +  A(0,3)*B(3,0)
ZA0.s[0][1] += A(0,0)*B(0,1)  +  A(0,1)*B(1,1)  +  A(0,2)*B(2,1)  +  A(0,3)*B(3,1)
ZA0.s[0][2] += A(0,0)*B(0,2)  +  A(0,1)*B(1,2)  +  A(0,2)*B(2,2)  +  A(0,3)*B(3,2)
ZA0.s[0][3] += A(0,0)*B(0,3)  +  A(0,1)*B(1,3)  +  A(0,2)*B(2,3)  +  A(0,3)*B(3,3)
```

These 32-bit results are then stored in the first horizontal slice of the ZA0 tile.


Note

This specific example assumes that the hardware implementation uses a 128-bit SVL, which gives an SVLs of 4. This means that each row in the ZA tile can contain four 32-bit values. Implementations that use a bigger SVL have bigger ZA tiles containing more results.

This widening `umopa` instruction requires matrix data to be prepared in advance. Specifically, matrices must be padded with zeros. During re-arrangement the output matrices are allocated with a modified K equal to `K_mod = K*ceil(K/4)`, that is a multiple of four. This avoids calculation errors that would result from using unknown values outside the matrix bounds.

The initial input matrices are stored in memory as row-major arrays. Matrix multiplication is performed as the sum of the outer product of one column from `matLeft` and one row from `matRight`.

The code rearranges data in both the `matLeft` and `matRight` matrices. This rearrangement is required because:

- A left matrix transposition is needed to implement a matrix multiplication with outer-products. This example uses a per-block matrix transposition.
- Four-way interleaving is required because the four-way `umopa` instruction performs the sum of four outer-products and accumulates the results.

The implementation therefore has the following steps:

1. Rearrange the `matRight` matrix, using the function `preprocess_r`.

   Row elements from the `matRight` matrix are four-way interleaved and contiguously stored to memory in blocks of 2 x SVLs columns. Each matrix row is zero-padded to 2 * SVLs elements. The final block of matrix columns is zero-padded to a multiple of 4 rows, if required. This rearranged data is called `matRight_mod`.
2. Rearrange the `matLeft` matrix, using the function `preprocess_l`.

   Elements from the `matLeft` matrix transpose-like re-arranged and contiguously stored to memory in blocks of SVLs rows x SVLs columns. Each block of SVLs rows is 32-bit width transposed and contiguously stored to memory. Each 32-bit container of the re-arranged matrix contains elements from 4 consecutive columns. This rearranged data is called `matLeft_mod`.
3. Multiply the rearranged `matLeft_mod` and `matRight_mod` matrices using the outer product instruction in the function `matmul_opt`.

   This function contains three nested loops:

   a. The outermost loops iterates over the rows (M) of the result matrix.
   b. The middle loop iterates over the columns (N) of the result matrix.
   c. The innermost loop iterates over the K dimension, producing result matrix elements as a sum of products.

The following sections describe these operations in more detail.

6.2 preprocess_r code

The following code shows the function `preprocess_r` with numbered lines. Subsequent sections explain how the code operates.

```
1.  preprocess_r:  // x0: K, x1: N, x2: matRight, x3: matRight_mod
2.      smstart sm              // Enable Streaming mode
3.
4.  // constants
5.      cntb    x5
```

```
6.      lsl     x16, x1, #1     // 2*ldb
7.      add     x10, x16, x1    // 3*ldb
8.      add     x4, x0, #3
9.      lsr     x4, x4, #2      // nbr_mod/4 = ceil(nbr/4)
10.     mul     x11, x4, x5     // (nbr_mod/4)*SVLb
11.     lsl     x17, x11, #1    // 2*(nbr_mod/4)*SVLb
12.     mov     x15, #0         // psel variable
13.     cnth    x13             // SVLb/2
14.
15.     ptrue   pn9.b
16.
17.     add     x8, x2, x1      // N dimension exit condition
18.     whilelt p2.b, x2, x8    // N dimension predicate
19.
20. .Loop_N:
21.     mov     x7, x2          // b_ptr
22.     mov     x9, x3          // b_mod_ptr
23.     whilelt p1.b, xzr, x0   // K dimension predicate
24.
25.     // Store predicates
26.     psel    pn11, pn9, p2.b[w15, 0]
27.     psel    pn12, pn9, p2.b[w13, 0]
28.
29.     mov     x6, xzr         // Loop_K counter
30. .Loop_K:
31.     psel    p0, p2, p1.b[w15, 0]
32.     psel    p3, p2, p1.b[w15, 1]
33.     ld1b    {z0.b}, p0/z, [x7]
34.     ld1b    {z1.b}, p3/z, [x7, x1]
35.
36.     psel    p0, p2, p1.b[w15, 2]
37.     psel    p3, p2, p1.b[w15, 3]
38.     ld1b    {z2.b}, p0/z, [x7, x16]
39.     ld1b    {z3.b}, p3/z, [x7, x10]
40.
41.     zip    { z8.b - z11.b }, { z0.b - z3.b } // 4-way interleave from 4 vectors
42.
43.     st1b    { z8.b-z9.b }, pn11, [x9]        // b_mod_ptr
44.     st1b    { z10.b-z11.b }, pn12, [x9, x17] // b_mod_ptr + 2*(nbr_mod/4)*SVLb
45.
46.     add     x7, x7, x1, lsl #2            // &b_ptr += 4*ldb
47.     addvl   x9, x9, #2                    // &b_mod_ptr += 2*SVLb
48.     add     x6, x6, #4                    // Loop_K counter increment
49.     whilelt p1.b, x6, x0                  // K dimension predicate
50.     b.first .Loop_K
51.
52.     add     x3, x3, x17, lsl #1           // &b_mod += 4*ceil(nbr/4)*SVLb
53.     addvl   x2, x2, #1                    // &b_base += SVLb
54.     whilelt p2.b, x2, x8                  // N dimension predicate
55.     b.first .Loop_N
56.
57.     smstop sm                             // Disable Streaming mode
58.
59.     ret
```

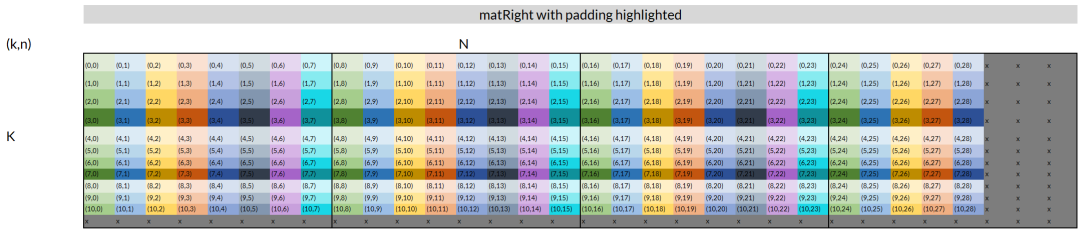
6.3 preprocess_r function overview

The `preprocess_r` function rearranges the `matRight` matrix in blocks of (2 * SVLs) columns. Each 32-bit element in the rearranged matrix data contains elements from four consecutive rows. Each matrix row is zero-padded to 2 * SVLs elements. The final block is padded to a multiple of 4 rows. The rearranged data is called `matRight_mod`.

The following example shows how the `preprocess_r` function rearranges the data.

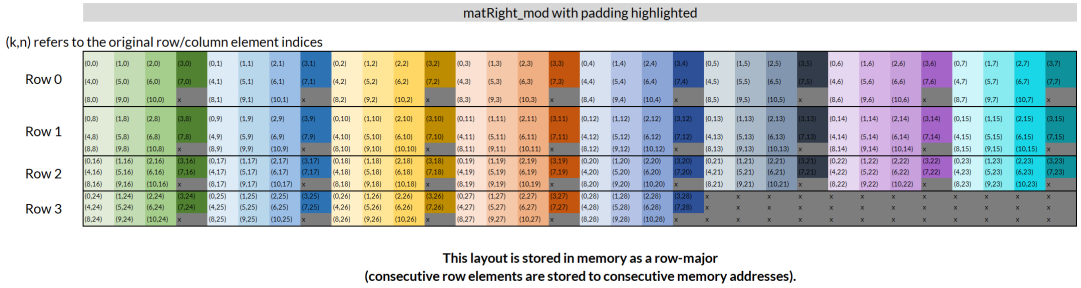
Consider the following `matRight` matrix with 11 rows (K) and 29 columns (N), padded to 12 rows (K) and 32 columns (N):

Figure 6-2: Example matRight matrix memory view



The `preprocess_r` function rearranges this example `matRight` matrix as shown in Figure 6-3: Example matRight memory layout after preprocess_r on page 76, assuming that SVL is 128b. In this example, four consecutive rows of four uint8 values are loaded into a single SME Z vector.

Figure 6-3: Example matRight memory layout after preprocess_r



The following sections examine specific portions of the code in more detail.

6.4 preprocess_r function details

This section describes how the `preprocess_r` function operates, looking at sections of the code in turn.

- On entry, function arguments are passed in registers as follows:
  - x0: K, the number of rows in the `matRight` matrix
  - x1: N, the number of columns in the `matRight` matrix
  - x2: The base address of the input matrix, `matRight`
  - x3: The base address of the rearranged matrix, `matRight_mod`
- Lines 33-34 and 38-39

  The `ld1b` instructions load four vectors from four consecutive matrix rows.

- Line 41

  The four vectors are four-way interleaved by the four-vector `zip` SME2 instruction.

- Lines 43-44

  The two-vector `st1b` instructions store the resulting four vectors as follows:

  - The first 2 vectors are consecutively stored to the first rearranged block, pointed to by `b_mod_ptr`.

    These vectors are consumed together first in `matmul_opt` for all K values, storing the first 2 x SVLs columns consecutively after rearrangement.

  - The second 2 vectors are consecutively stored to the second rearranged block. This block is at an offset of 2 x (K_mod / 4) x SVLb, which is equal to 2 x SVLs x K_mod (K_mod = ceil(K/4) * 4).

    These vectors are consumed by the next N dimension loop iteration in `matmul_opt`. This is why they are stored with an offset.

6.5 preprocess_l code

The following code shows the `preprocess_l` function with numbered lines. Subsequent sections explain how the code operates.

```
1.  preprocess_l:   // x0: M, x1: K, x2: matLeft, x3: matLeft_mod
2.    stp     x19, x20, [sp, #-32]!
3.
4.    smstart                         // Enable Streaming mode & ZA
5.
6.  // constants
7.    cntw    x5                      // SVLs
8.    mul     x11, x5, x1             // SVLs*nbc
9.    add     x18, x1, #3
10.   lsr     x18, x18, #2            // ceil(nbc/4)
11.   mul     x15, x18, x5            // SVLs*ceil(nbc/4)
12.   lsl     x18, x15, #2            // SVLs*ceil(nbc/4)*4
13.
14.   mul     x4, x5, x5              // SVLs*SVLs
15.   lsl     x16, x4, #1             // 2*SVLs*SVLs
16.   add     x16, x16, x4            // 3*SVLs*SVLs
17.   cntb    x17                     // SVLb
18.
19.   mov     x8, #0                  // Loop_M counter
20.   whilelt p0.s, x8, x0            // M dimension predicate
21.
22. .Loop_M:
23.   mov     x7, x3                  // a_mod_base
24.   mov     x10, x2                 // a_base
25.   add     x9, x2, x1              // Loop_K exit condition
26.   whilelt pn12.b, x2, x9, vlx4    // K dimension predicate-as-counter
27.   mov     x13, #0                 // offset0=0
28.   mov     x14, x4                 // offset1=SVLs*SVLs
29.   lsl     x19, x4, #1             // offset2=2*SVLs*SVLs
30.   mov     x20, x16                // offset3=3*SVLs*SVLs
31.
32. .Loop_K:
33.   mov     x6, x10                 // a_ptr
```

```
34.
35.   mov     w12, #0                 // Loop_load counter
36. .Loop_load:
37.   psel    pn8, pn12, p0.b[w12, #0]
38.   psel    pn9, pn12, p0.b[w12, #4]
39.   ld1b    {z0.b-z3.b}, pn8/z, [x6]      // Load 4 vectors from a_ptr
40.   ld1b    {z4.b-z7.b}, pn9/z, [x6, x1]  // Load 4 vectors from a_ptr + nbc
41.   mova    za0h.b[w12, 0:3], {z0.b-z3.b} // za0h.s, za1h.s, za2h.s, za3h.s: row 1
42.   mova    za0h.b[w12, 4:7], {z4.b-z7.b} // za0h.s, za1h.s, za2h.s, za3h.s: row 2
43.   add     w12, w12, #8                  // Loop_load counter increment
44.   add     x6, x6, x1, lsl #1            // a_ptr += 2*nbc INT8 elems
45.   cmp     w12, w17
46.   b.mi    .Loop_load
47.
48.   mov     w12, #0                         // Loop_store counter
49. .Loop_store:
50.   whilelt pn8.s, x13, x15, vlx4           // Tile0 store predicate-as-counter
51.   whilelt pn9.s, x14, x15, vlx4           // Tile1 store predicate-as-counter
52.   whilelt pn10.s, x19, x15, vlx4          // Tile2 store predicate-as-counter
53.   whilelt pn11.s, x20, x15, vlx4          // Tile3 store predicate-as-counter
54.   mova    {z0.s-z3.s}, za0v.s[w12, 0:3]
55.   mova    {z4.s-z7.s}, za1v.s[w12, 0:3]
56.   mova    {z8.s-z11.s}, za2v.s[w12, 0:3]
57.   mova    {z12.s-z15.s}, za3v.s[w12, 0:3]
58.   add     w12, w12, #4                        //Inc Loop_store counter
59.   st1w    {z0.s-z3.s},pn8,[x7, x13, lsl #2]   //1st 4 cols Tile0: a_mod+offset0
60.   st1w    {z4.s-z7.s},pn9,[x7, x14, lsl #2]   //1st 4 cols Tile1: a_mod+offset1
61.   st1w    {z8.s-z11.s},pn10,[x7, x19, lsl #2] //1st 4 cols Tile2: a_mod+offset2
62.   st1w    {z12.s-z15.s},pn11,[x7, x20, lsl #2]//1st 4 cols Tile3: a_mod+offset3
63.   incw    x13, all, mul #4           // Tile0 store pointer offset increment
64.   incw    x14, all, mul #4           // Tile1 store pointer offset increment
65.   incw    x19, all, mul #4           // Tile2 store pointer offset increment
66.   incw    x20, all, mul #4           // Tile3 store pointer offset increment
67.   cmp     w12, w5
68.   b.mi    .Loop_store
69.
70.   add     x13, x13, x16         // Tile0 store pointer offset += 3*SVLs*SVLs
71.   add     x14, x14, x16         // Tile1 store pointer offset += 3*SVLs*SVLs
72.   add     x19, x19, x16         // Tile2 store pointer offset += 3*SVLs*SVLs
73.   add     x20, x20, x16         // Tile3 store pointer offset += 3*SVLs*SVLs
74.   addvl   x10, x10, #4          // a_base += 4*SVLb INT8 elems
75.   whilelt pn12.b, x10, x9, vlx4 // K dimension predicate-as-counter
76.   b.first .Loop_K
77.
78.   add     x2, x2, x11           // &a += SVLs*nbc INT8 elems
79.   add     x3, x3, x18           // &a_mod += SVLs*ceil(nbc/4)*4 INT8 elems
80.
81.   incw    x8
82.   whilelt p0.s, x8, x0
83.   b.first .Loop_M
84.
85.   smstop                         // Disable Streaming mode & ZA
86.
87.   ldp     x19, x20, [sp], #32
88.
89.   ret
```

6.6 preprocess_l function overview

The `preprocess_l` function rearranges the `matLeft` matrix, so that blocks of SVLs rows x SVLb columns from the `matLeft` matrix are transposed with 32-bit transpose width, taking 4 consecutive 8-bit elements from each row, and contiguously stored to memory.

Each 32-bit element in the rearranged matrix data contains elements from four consecutive columns. Each matrix row is zero-padded to a multiple of four elements. The final block of matrix rows is zero-padded to SVLs rows, if required. This rearranged data is called `matLeft_mod`.

The following example shows how the `preprocess_l` function rearranges data.

Consider the following `matLeft` matrix with 7 rows (M) and 6 columns (K):

Figure 6-4: Example matLeft matrix

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)

The layout of `matLeft_mod` in memory after calling `preprocess_l` is shown in Figure 6-5: Example matLeft memory layout after preprocess_l on page 80, with 32 bits or 4 bytes of memory per row. In this example, four consecutive rows of four uint8 values are loaded into a single SME Z vector. This example assumes SVL is 128b.

Figure 6-5: Example matLeft memory layout after preprocess_l

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)
(0,4)	(0,5)	0	0
(1,4)	(1,5)	0	0
(2,4)	(2,5)	0	0
(3,4)	(3,5)	0	0
(4,0)	(4,1)	(4,2)	(4,3)
(5,0)	(5,1)	(5,2)	(5,3)
(6,0)	(6,1)	(6,2)	(6,3)
0	0	0	0
(4,4)	(4,5)	0	0
(5,4)	(5,5)	0	0
(6,4)	(6,5)	0	0
0	0	0	0

The preprocess_l function details section describes specific portions of the code in more detail.

6.7 preprocess_l function details

This section describes how the `preprocess_l` function operates, looking at sections of the code in turn.

- On entry, function arguments are passed in registers as follows:
  - x0: M, the number of rows in the `matLeft` matrix
  - x1: K, the number of columns in the `matLeft` matrix
  - x2: The base address of the input matrix, `matLeft`
  - x3: The base address of the rearranged matrix, `matLeft_mod`
- Lines 39-40:

Each 4-vector `ld1b` instruction loads four vectors from a single row of the input matrix data. Each block transposition uses four vector loads and four 32-bit ZA tiles.

- Lines 41-42:

  The `mova` instructions move the data from Z vectors to horizontal slices of 8-bit element ZA tiles, as follows:

```
mova    za0h.b[w12, 0:3], { z0.b-z3.b }

ZA0h.s[0] = ZA0h.b[0] = Z0
ZA1h.s[0] = ZA0h.b[1] = Z1
ZA2h.s[0] = ZA0h.b[2] = Z2
ZA3h.s[0] = ZA0h.b[3] = Z3
```

  This code loads horizontal slices of 8-bit ZA tiles, but stores vertical slices of 32-bit ZA tiles. This leverages 4-vector length loads from memory and places them each in its own tile.

- Lines 49-68 (`Store_loop`)

  `Store_loop` stores elements from 32-bit ZA tiles to memory. Each iteration stores 4 vertical slices of a 32-bit ZA tile into consecutive memory locations, using predication to determine when the bounds of the array are exceeded.

  The `whilelt` instructions in lines 50-53 generate predicate-as-counters for four-vector consecutive stores. The `vlx4` in these instructions is the vector length specifier, and indicates that the predicate can control four vectors. The predicates ensure that non-existent columns in the ZA tiles are not stored to memory, using the base address to calculate the store bounds for each of the ZA tiles.

  The `mova` instructions in lines 54-57 move four vertical 32-bit slices of ZA0-ZA3 to consecutive groups of Z vectors. The `v` suffix accesses the data in the ZA tile as columns, rearranging the matrix.

  The `st1w` instructions in lines 59-62 perform vector contiguous stores from the Z vectors to memory.

6.8 matmul_opt code

The following code shows the `matmul_opt` function with numbered lines.

```
1.   matmul_opt:
2.   // x0: M, x1: K, x2: N, x3: matLeft_mod, x4: matRight_mod, x5: matResult_opt
3.     stp     x19, x20, [sp, #-48]!
4.     stp     x21, x22, [sp, #16]
5.     str     x23, [sp, #32]
6.
7.     smstart                         // Enable Streaming mode & ZA
8.
9.   // constants
10.    cntb    x6                      // SVLb
11.    cntw    x15                     // SVLs
12.    lsl     x11, x2, #2             // 4*N
```

```
13.    mul     x21, x15, x2            // SVLs*ldc
14.    add     x18, x21, x2            // (SVLs+1)*ldc
15.    add     x7, x1, #3
16.    lsr     x7, x7, #2              // ceil(K/4)
17.    mul     x7, x7, x6              // ceil(K/4)*SVLb
18.    lsl     x0, x0, #2              // 4*M
19.    mov     x12, #0                 // Loop_M counter
20.    mov     x15, #0                 // psel variable
21.    sub     w6, w6, #8              // SVLb-8
22.    ptrue   pn10.b                  // Predicate for SME2 VLx2 (a_ptr loads)
23.    whilelt p2.b, x12, x0           // Tile 0/1 predicate (M dimension)
24.
25.  .Loop_M:
26.    addvl   x12, x12, #1            // Loop_M counter increment
27.    whilelt p3.b, x12, x0           // Tile 2/3 predicate (M dimension)
28.
29.    mov     x19, x4                 // b_base
30.    mov     x22, x5                 // c_base
31.    mov     x13, #0                 // Loop_N counter
32.    add     x10, x3, x7             // a_base + 4*ceil(K/4)*SVLs
33.    add     x17, x3, x7             // matLeft row0 end address
34.    addvl   x9, x17, #-1            // Loop_K exit condition
35.
36.
37.  .Loop_N:
38.    mov     x8, x3                  // a_ptr = a_base
39.    mov     x20, x19                // b_ptr = b_base
40.    mov     x23, x22                // c_ptr = c_base
41.
42.    pext    { p0.b, p1.b }, pn9[0]  // Tile 0/2 and tile 1/3 predicates
43.
44.    zero    {za}
45.
46.    ld1b    {z1.b}, p2/z, [x8]      // Load 1st vector from a_ptr
47.
48.    whilelt pn10.b, x8, x17, vlx2     // K dimension predicate-as-counter
49.    ld1b    {z2.b-z3.b}, pn9/z, [x20] // Load 2 vectors from b_ptr
50.
51.    umopa   za0.s, p2/m, p0/m, z1.b, z2.b  // ZA0 += 1st a_ptr OP 1st b_ptr
52.    ld1b    {z5.b}, p3/z, [x8, x7]  // Load 2nd vec from a_ptr+ceil(K/4)*SVLb
53.    addvl   x8, x8, #1              // a_ptr += SVLb
54.
55.  .Loop_K:
56.    umopa   za2.s, p3/m, p0/m, z5.b, z2.b // ZA2 += 2nd a_ptr OP 1st b_ptr
57.    umopa   za1.s, p2/m, p1/m, z1.b, z3.b // ZA1 += 1st a_ptr OP 2nd b_ptr
58.    psel    pn11, pn10, p3.s[w15, #0]
59.    ld1b    {z0.b-z1.b}, pn10/z, [x8]     // Load 2 vectors from a_ptr
60.    umopa   za3.s, p3/m, p1/m, z5.b, z3.b // ZA3 += 2nd a_ptr OP 2nd b_ptr
61.    ld1b    {z6.b-z7.b}, pn9/z, [x20, #2, mul vl] // 2 vecs from b_ptr+2*SVLb
62.
63.    umopa   za0.s, p2/m, p0/m, z0.b, z6.b // ZA0 += 1st a_ptr OP 1st b_ptr
64.    ld1b   {z4.b-z5.b}, pn11/z, [x8, x7]  // 2 vecs from a_ptr+ceil(K/4)*SVLb
65.
66.    umopa   za2.s, p3/m, p0/m, z4.b, z6.b // ZA2 += 2nd a_ptr OP 1st b_ptr
67.    addvl   x20, x20, #4                  // b_ptr += 4*SVLb
68.
69.    umopa   za1.s, p2/m, p1/m, z0.b, z7.b // ZA1 += 1st a_ptr OP 2nd b_ptr
70.
71.    umopa   za3.s, p3/m, p1/m, z4.b, z7.b // ZA3 += 2nd a_ptr OP 2nd b_ptr
72.    ld1b    {z2.b-z3.b}, pn9/z, [x20]     // Load 2 vectors from b_ptr
73.
74.    umopa   za0.s, p2/m, p0/m, z1.b, z2.b // ZA0 += 1st a_ptr OP 1st b_ptr
75.    addvl   x8, x8, #2                    // a_ptr += 2*SVLb
76.
77.
78.    cmp     x8, x9
79.    b.mi    .Loop_K
80.
81.    umopa   za2.s, p3/m, p0/m, z5.b, z2.b // ZA2 += 2nd a_ptr OP 1st b_ptr
82.
83.    umopa   za1.s, p2/m, p1/m, z1.b, z3.b // ZA1 += 1st a_ptr OP 2nd b_ptr
```

```
84.
85.    umopa   za3.s, p3/m, p1/m, z5.b, z3.b
86.    addvl   x20, x20, #2                  // b_ptr += 2*SVLb
87.
88.    cmp     x8, x10
89.    b.ge    .Ktail_end
90.
91.  .Ktail_start:
92.    ld1b    {z1.b}, p2/z, [x8]
93.
94.    ld1b    {z2.b-z3.b}, pn9/z, [x20]
95.
96.    umopa   za0.s, p2/m, p0/m, z1.b, z2.b
97.    ld1b    {z14.b}, p3/z, [x8, x7]
98.
99.    umopa   za2.s, p3/m, p0/m, z14.b, z2.b
100.
101.   umopa   za1.s, p2/m, p1/m, z1.b, z3.b
102.   addvl   x20, x20, #2                  // b_ptr += 2*SVLb
103.
104.   umopa   za3.s, p3/m, p1/m, z14.b, z3.b
105.
106. .Ktail_end:
107.   // store results
108.   mov     w14, #0                       // Loop_store_ZA counter
109.   psel    pn8, pn9, p2.b[w14, 0]
110.   psel    pn11, pn9, p3.b[w14, 0]
111. // ZA tiles to vecs: z0=za0h.s[0], z1= za1h.s[0], z2=za2h.s[0], z3=za3h.s[0]
112.   mova    { z0.b-z3.b }, za0h.b[w14, 0:3]
113.   st1w    { z0.s-z1.s }, pn8, [x23]               // Store to c_ptr
114.   st1w    { z2.s-z3.s }, pn11, [x23, x21, lsl #2] // Store to c_ptr+SVLs*ldc
115. .Loop_store_ZA:
116.   psel    pn8, pn9, p2.b[w14, 4]
117.   psel    pn11, pn9, p3.b[w14, 4]
118.   mova    { z0.b-z3.b }, za0h.b[w14, 4:7]
119.   st1w    { z0.s-z1.s }, pn8, [x23, x2, lsl #2]   // to c_ptr+ldc
120.   st1w    { z2.s-z3.s }, pn11, [x23, x18, lsl #2] // to c_ptr+(SVLs+1)*ldc
121.
122.   add     x23, x23, x2, lsl #3          // c_ptr += 2*ldc INT32 elms
123.   add     w14, w14, #8                  // Loop_store_ZA counter increment
124.
125.   psel    pn8, pn9, p2.b[w14, 0]
126.   psel    pn11, pn9, p3.b[w14, 0]
127.   mova    { z0.b-z3.b }, za0h.b[w14, 0:3]
128.   st1w    { z0.s-z1.s }, pn8, [x23]               // Store to c_ptr
129.   st1w    { z2.s-z3.s }, pn11, [x23, x21, lsl #2] // Store to c_ptr+SVLs*ldc
130.   cmp     w14, w6
131.   b.mi    .Loop_store_ZA
132.
133.   psel    pn8, pn9, p2.b[w14, 4]
134.   psel    pn11, pn9, p3.b[w14, 4]
135.   mova    { z0.b-z3.b }, za0h.b[w14, 4:7]
136.   st1w    { z0.s-z1.s }, pn8, [x23, x2, lsl #2]   // to c_ptr+ldc
137.   st1w    { z2.s-z3.s }, pn11, [x23, x18, lsl #2] // to c_ptr+(SVLs+1)*ldc
138.
139.   addvl   x22, x22, #2            // &c_base += 2*SVLb
140.   addvl   x13, x13, #2            // Loop_N counter increment
141.   whilelt pn9.b, x13, x11, vlx2   // Tiles predicate-as-counter (N dimension)
142.   add     x19, x19, x7, lsl #1    // &b_base += 2*SVLs*4*ceil(K/4)
143.   b.first .Loop_N
144.
145.   add     x3, x3, x7, lsl #1      // a_base += 2*SVLs*4*ceil(K/4)
146.   add     x5, x5, x21, lsl #3     // c_base += 2*SVLs*ldc INT32 elms
147.   addvl   x12, x12, #1            // Loop_M counter increment
148.   whilelt p2.b, x12, x0           // Tile 0/1 predicate (M dimension)
149.   b.first    .Loop_M
150.
151.   smstop                          // Disable Streaming mode & ZA
152.
153.   ldr     x23, [sp, #32]
154.   ldp     x21, x22, [sp, #16]
```

```
155.    ldp     x19, x20, [sp], #48
156.
157.    ret
```

6.9 matmul_opt function overview

The `matmul_opt` function inner loop iterates over the rearranged columns of the `matLeft` matrix (`matLeft_mod` buffer) and rearranged rows of the `matRight` matrix (`matRight_mod` buffer). The outer products are calculated and stored in `matResult_opt`.

Results are accumulated into four 32-bit ZA tiles, with 2x2 tiling.

The `matmul_opt` function in this `matmul_int8` example behaves in the same way as the `matmul_fp32` example, but using `umopa` instructions rather than `fmopa` instructions.

See matmul_fp32 for details.

7. gemv_cm_int8: 8-bit integer to 32-bit integer matrix-by-vector multiplication

The `gemv_cm_int8` example implements generalized matrix-by-vector multiplication (gemv), with column-major unsigned 8-bit integer inputs and unsigned 32-bit integer outputs.

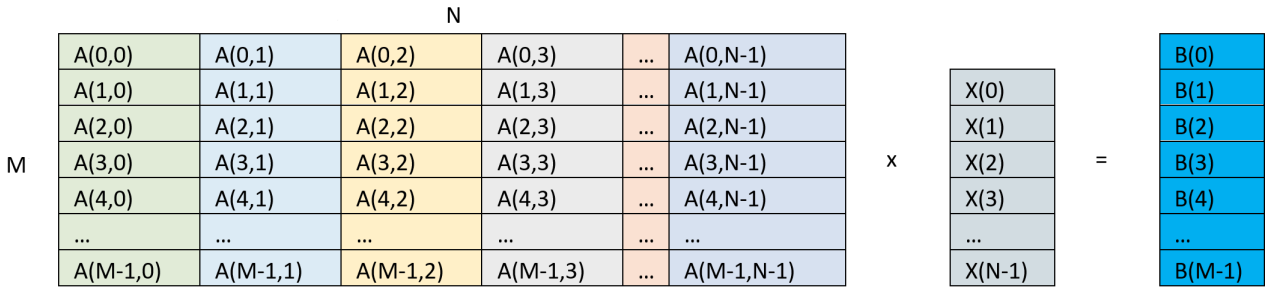
7.1 Overview of the gemv_cm_int8 algorithm

The `gemv_cm_int8` example multiplies a matrix A by a vector X, to produce a result vector B.

- `A` is an M x N input matrix in column-major format containing unsigned 8-bit integer values
- `X` is an N x 1 input vector containing unsigned 8-bit integer values
- `B` is an M x 1 result vector containing unsigned 32-bit integer values

Figure 7-1: The gemv operation on page 85 shows the gemv operation:

Figure 7-1: The gemv operation

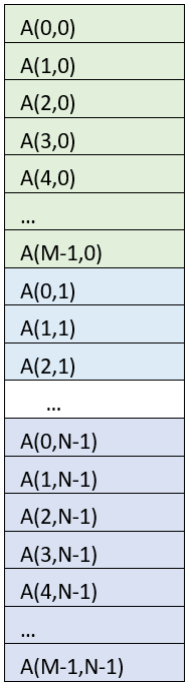


Each element `B(i)` in the result vector `B` is the result of the following calculation:

```
B(i) =     A(i,0)*X(0)
         + A(i,1)*X(1)
         + A(i,2)*X(2)
         + A(i,3)*X(3)
         + A(i,4)*X(4)
         + ...
         + A(i,N-1)*X(N-1)
```

Figure 7-2: Memory layout of the input matrix on page 86 shows the memory layout of the input matrix `A` in memory, in column-major format with each unsigned 8-bit value occupying a single byte in memory:

Figure 7-2: Memory layout of the input matrix



7.2 gemv_opt code

The following code shows the `gemv_opt` function with numbered lines. Subsequent sections explain how the code operates.

Note

For clarity, the code example consists of a function `gemv_opt` function which calls a macro `col_loop`. The code below shows the `col_loop` macro first, followed by the `gemv_opt` function.

```
1.   // Input parameters
2.   #define A         x0 // Column-major matrix 'A' (const uint8_t *)
3.   #define x         x1 // Vector 'x' (const uint8_t *)
4.   #define B         x2 // Output vector 'B = Ax' (uint32_t *B)
5.   #define a_rows    x3 // Number of rows in 'A' (uint64_t)
6.   #define a_cols    x4 // Number of cols in 'A' (uint64_t)
7.
8.   // Working registers
9.   #define A_col_ptr       x5   // Pointer to iterate through columns of 'A'
10.  #define A_col_start_ptr x6   // Base pointer used to set 'A_col_ptr'
11.  #define x_row_ptr       x7   // Pointer to iterate through elements of 'x'
12.  #define x_end_ptr       x8   // End of the input vector 'x'
13.  #define za_elems        x9   // Number of words in ZA (4 * cntw * cntw)
14.  #define a_row_ix        x10  // Outer row loop counter for rows of 'A'
15.  #define row_base        w11  // Row group index (ZA vector select)
16.  #define psel_base_ix    w12  // Base index for PSEL (always zero)
17.  #define row_offset      x13  // Inner row loop counter for rows of 'A'
18.  #define row_ix_target   x14  // Index target for 'a_row_ix' and 'row_offset'
19.                              // to reach this iteration (# of rows to process)
```

```
20.  #define a_col_stride  a_rows // 'A' column stride
21.  #define a_col_stride_2x  x15 // 'A' two column stride
22.  #define a_col_stride_3x  x16 // 'A' three column stride
23.
24.  // Predicates
25.  #define row_pred            p0 // Byte predicate for the K dimension
26.  #define row_pred_count_mask p1 // Byte predicate with first granule all active
27.                                 // aside from its very first element
28.  #define tmp_pred            p2 // Temporary predicate register
29.  #define col_pred_n   pn8  // Predicate-as-counter for column loads/stores
30.  #define col_pred_p   p8   // (Above in predicate-as-mask form)
31.  #define col1_pred_n pn9  // Combined predicate for the first of 4 col loads
32.  #define col1_pred_p p9   // (Above in predicate-as-mask form)
33.  #define col2_pred_n pn10 // Combined predicate for the second of 4 col loads
34.  #define col2_pred_p p10  // (Above in predicate-as-mask form)
35.  #define col3_pred_n pn11 // Combined predicate for the third of 4 col loads
36.  #define col3_pred_p p11  // (Above in predicate-as-mask form)
37.  #define col4_pred_n pn12 // Combined predicate for the last of 4 col loads
38.  #define col4_pred_p p12  // (Above in predicate-as-mask form)
39.
40.  // Column loop: iterate through columns of A in groups of sixteen columns.
41.  // Accumulate results for the current set of rows up to row_ix_target in ZA
42.  .macro col_loop
43.    whilelt row_pred.b, x_row_ptr, x_end_ptr
44.    b.none ._col_loop_end
45.    ._col_loop:
46.      // Load-replicate up to sixteen 8-bit elements from 'x'
47.      ld1rqb {z15.b}, row_pred/z, [x_row_ptr]
48.
49.      // Inner row loop: iterate through elements in the current sixteen
50.      // columns (associated with the loaded data from 'x') in 'row groups'
51.      // of 4*SVL elements up to `row_ix_target`, using UVDOT to accumulate
52.      // results into ZA.
53.      mov row_base, #0
54.      mov row_offset, a_row_ix
55.      whilelt col_pred_n.b, a_row_ix, a_rows, vlx4
56.      ._inner_row_loop:
57.        // Get the pointer to the relevant part of the first column in the
58.        // current group of sixteen for this row group
59.        add A_col_ptr, A_col_start_ptr, row_offset
60.
61.        // Iteration #1: process first four columns for this row group
62.          // Combine row and column predicates to avoid loading beyond
63.          // the end of the matrix (no awkward tail loops required!)
64.          psel col1_pred_p, col_pred_p, row_pred.b[psel_base_ix, 0]
65.          psel col2_pred_p, col_pred_p, row_pred.b[psel_base_ix, 1]
66.          psel col3_pred_p, col_pred_p, row_pred.b[psel_base_ix, 2]
67.          psel col4_pred_p, col_pred_p, row_pred.b[psel_base_ix, 3]
68.
69.          // Load 4*SVLb elements from each of four columns in 'A'
70.          ld1b {z16.b, z20.b, z24.b, z28.b}, col1_pred_n/z, [A_col_ptr]
71.          ld1b {z17.b, z21.b, z25.b, z29.b}, col2_pred_n/z, [A_col_ptr,
                     a_col_stride]
72.          ld1b {z18.b, z22.b, z26.b, z30.b}, col3_pred_n/z, [A_col_ptr,
                     a_col_stride_2x]
73.          ld1b {z19.b, z23.b, z27.b, z31.b}, col4_pred_n/z, [A_col_ptr,
                     a_col_stride_3x]
74.
75.          // Take the 'vertical' dot product between the loaded elements of 'A'
76.          // and elements of 'x', multiplying elements in rows of 'A' by columns
77.          // of 'x' and reducing along the K dimension (accumulating widened
78.          // 8-bit -> 32-bit results in ZA).
79.          uvdot za.s[row_base, 0, vgx4], {z16.b - z19.b}, z15.b[0]
80.          uvdot za.s[row_base, 1, vgx4], {z20.b - z23.b}, z15.b[0]
81.          uvdot za.s[row_base, 2, vgx4], {z24.b - z27.b}, z15.b[0]
82.          uvdot za.s[row_base, 3, vgx4], {z28.b - z31.b}, z15.b[0]
83.
84.          // A_col_ptr += 4 cols
85.          add A_col_ptr, A_col_ptr, a_col_stride, lsl #2
86.
87.        // Iteration #2: process second four columns for this row group
```

```
88.           // Combine row and column predicates to avoid loading beyond
89.           // the end of the matrix (no awkward tail loops required!)
90.           psel col1_pred_p, col_pred_p, row_pred.b[psel_base_ix, 4]
91.           psel col2_pred_p, col_pred_p, row_pred.b[psel_base_ix, 5]
92.           psel col3_pred_p, col_pred_p, row_pred.b[psel_base_ix, 6]
93.           psel col4_pred_p, col_pred_p, row_pred.b[psel_base_ix, 7]
94.
95.           // Load 4*SVL elements from each of four columns in 'A'
96.           ld1b {z16.b, z20.b, z24.b, z28.b}, col1_pred_n/z, [A_col_ptr]
97.           ld1b {z17.b, z21.b, z25.b, z29.b}, col2_pred_n/z, [A_col_ptr,
                      a_col_stride]
98.           ld1b {z18.b, z22.b, z26.b, z30.b}, col3_pred_n/z, [A_col_ptr,
                      a_col_stride_2x]
99.           ld1b {z19.b, z23.b, z27.b, z31.b}, col4_pred_n/z, [A_col_ptr,
                      a_col_stride_3x]
100.
101.          // Take the vertical dot product, much as in iteration #1 but
102.          // selecting the second set of four 8-bit elements from z20
103.          uvdot za.s[row_base, 0, vgx4], {z16.b - z19.b}, z15.b[1]
104.          uvdot za.s[row_base, 1, vgx4], {z20.b - z23.b}, z15.b[1]
105.          uvdot za.s[row_base, 2, vgx4], {z24.b - z27.b}, z15.b[1]
106.          uvdot za.s[row_base, 3, vgx4], {z28.b - z31.b}, z15.b[1]
107.
108.          // A_col_ptr += 4 cols
109.          add A_col_ptr, A_col_ptr, a_col_stride, lsl #2
110.
111.        // Iteration #3: process third four columns for this row group
112.          // Combine row and column predicates to avoid loading beyond
113.          // the end of the matrix (no awkward tail loops required!)
114.          psel col1_pred_p, col_pred_p, row_pred.b[psel_base_ix,  8]
115.          psel col2_pred_p, col_pred_p, row_pred.b[psel_base_ix,  9]
116.          psel col3_pred_p, col_pred_p, row_pred.b[psel_base_ix, 10]
117.          psel col4_pred_p, col_pred_p, row_pred.b[psel_base_ix, 11]
118.
119.          // Load 4*SVL elements from each of four columns in 'A'
120.          ld1b {z16.b, z20.b, z24.b, z28.b}, col1_pred_n/z, [A_col_ptr]
121.          ld1b {z17.b, z21.b, z25.b, z29.b}, col2_pred_n/z, [A_col_ptr,
                      a_col_stride]
122.          ld1b {z18.b, z22.b, z26.b, z30.b}, col3_pred_n/z, [A_col_ptr,
                      a_col_stride_2x]
123.          ld1b {z19.b, z23.b, z27.b, z31.b}, col4_pred_n/z, [A_col_ptr,
                      a_col_stride_3x]
124.
125.          // Take the vertical dot product, much as in iteration #1 but
126.          // selecting the third set of four 8-bit elements from z20
127.          uvdot za.s[row_base, 0, vgx4], {z16.b - z19.b}, z15.b[2]
128.          uvdot za.s[row_base, 1, vgx4], {z20.b - z23.b}, z15.b[2]
129.          uvdot za.s[row_base, 2, vgx4], {z24.b - z27.b}, z15.b[2]
130.          uvdot za.s[row_base, 3, vgx4], {z28.b - z31.b}, z15.b[2]
131.
132.          // A_col_ptr += 4 cols
133.          add A_col_ptr, A_col_ptr, a_col_stride, lsl #2
134.
135.        // Iteration #4: process final four columns for this row group
136.          // Combine row and column predicates to avoid loading beyond
137.          // the end of the matrix (no awkward tail loops required!)
138.          psel col1_pred_p, col_pred_p, row_pred.b[psel_base_ix, 12]
139.          psel col2_pred_p, col_pred_p, row_pred.b[psel_base_ix, 13]
140.          psel col3_pred_p, col_pred_p, row_pred.b[psel_base_ix, 14]
141.          psel col4_pred_p, col_pred_p, row_pred.b[psel_base_ix, 15]
142.
143.          // Load 4*SVL elements from each of four columns in 'A'
144.          ld1b {z16.b, z20.b, z24.b, z28.b}, col1_pred_n/z, [A_col_ptr]
145.          ld1b {z17.b, z21.b, z25.b, z29.b}, col2_pred_n/z, [A_col_ptr,
                      a_col_stride]
146.          ld1b {z18.b, z22.b, z26.b, z30.b}, col3_pred_n/z, [A_col_ptr,
                      a_col_stride_2x]
147.          ld1b {z19.b, z23.b, z27.b, z31.b}, col4_pred_n/z, [A_col_ptr,
                      a_col_stride_3x]
148.
149.          // Take the vertical dot product, much as in iteration #1 but
```

```
150.         // selecting the final set of four 8-bit elements from z20
151.         uvdot za.s[row_base, 0, vgx4], {z16.b - z19.b}, z15.b[3]
152.         uvdot za.s[row_base, 1, vgx4], {z20.b - z23.b}, z15.b[3]
153.         uvdot za.s[row_base, 2, vgx4], {z24.b - z27.b}, z15.b[3]
154.         uvdot za.s[row_base, 3, vgx4], {z28.b - z31.b}, z15.b[3]
155.
156.       // Update inner row loop variables
157.       add row_base, row_base, #4       // row_base   += 1 row group
158.       addvl row_offset, row_offset, #4 // row_offset += 4 * SVL
159.       whilelt col_pred_n.b, row_offset, row_ix_target, vlx4 // Update col pred
160.       b.first ._inner_row_loop
161.
162.     // Update column loop variables
163.     add A_col_start_ptr, A_col_start_ptr, a_col_stride, lsl #4 // += 16 cols
164.     add x_row_ptr, x_row_ptr, #16 // x_row_ptr += 16 elems
165.     whilelt row_pred.b, x_row_ptr, x_end_ptr
166.     b.first ._col_loop
167.   ._col_loop_end:
168. .endm
169.
170. // void gemv_opt(const uint8_t *A, const uint8_t *x, uint32_t *B,
171. //               uint64_t a_rows, uint64_t a_cols);
172. .text
173. .align 2
174. .global gemv_opt
175. .type   gemv_opt, %function
176. gemv_opt:
177.   // Spill callee-saved registers
178.   stp d8,  d9,  [sp, #-64]!
179.   stp d10, d11, [sp, #16]
180.   stp d12, d13, [sp, #32]
181.   stp d14, d15, [sp, #48]
182.
183.   // Enter streaming mode
184.   smstart
185.
186.   // Bail out early for inputs with no columns
187.   cmp a_cols, #0
188.   b.le ._end
189.
190.   // Initialise constants
191.   mov row_base, #0
192.   mov psel_base_ix, #0
193.   cnth za_elems
194.   mul za_elems, za_elems, za_elems
195.   mov row_ix_target, za_elems
196.   add x_end_ptr, x, a_cols
197.   lsl a_col_stride_2x, a_col_stride, #1
198.   add a_col_stride_3x, a_col_stride, a_col_stride, lsl #1
199.
200.   // Outer row loop: iterate through rows of A in chunks of `za_elems` (up to
201.   // a_row_ix = `row_ix_target`), aiming to accumulate as many result
202.   // elements in B at each iteration as possible
203.   mov a_row_ix, #0
204.   ._row_loop:
205.     // Zero all tiles (prepare to accumulate new results)
206.     zero {za}
207.
208.     // Calculate the number of rows to process this iteration
209.     cmp row_ix_target, a_rows
210.     csel row_ix_target, row_ix_target, a_rows, lt
211.
212.     // Initialise variables for the inner loop
213.     mov x_row_ptr, x       // Reset x pointer (K = 0)
214.     mov A_col_start_ptr, A // Reset A pointer (K = 0)
215.
216.     // Column loop: iterate through the columns of A, accumulating results
217.     // for the current set of rows up to `row_ix_target` in Z, processing
218.     // in chunks of 16 columns at a time
219.     col_loop
220.
```

```
221.      // Store loop: iterate through the current set of rows to process up to
222.      // `row_ix_target` and store out results to B
223.      mov row_base, #0
224.      whilelt col_pred_n.s, a_row_ix, a_rows, vlx4
225.      ._store_loop:
226.        // Move results out of ZA and into Z
227.        mova {z0.d - z3.d}, za.d[row_base, 0, vgx4]
228.
229.        // Deinterleave results (arrange contiguously in vectors accounting
230.        // for the output format of UVDOT)
231.        zip {z0.s - z3.s}, {z0.s - z3.s}
232.
233.        // Store
234.        st1w {z0.s - z3.s}, col_pred_n, [B, a_row_ix, lsl #2]
235.
236.        // Update store loop variables
237.        add row_base, row_base, #1   // row_base += 1
238.        addvl a_row_ix, a_row_ix, #1 // a_row_ix += 4 * SVLs
239.        whilelt col_pred_n.s, a_row_ix, row_ix_target, vlx4 // Update col pred
240.        b.first ._store_loop
241.
242.      // Update outer row loop variables
243.      add row_ix_target, row_ix_target, za_elems
244.      cmp a_row_ix, a_rows
245.      b.lt ._row_loop
246.
247.    // Return
248.    ._end:
249.      // Exit streaming mode
250.      smstop
251.
252.      // Restore callee-saved registers and return
253.      ldp d14, d15, [sp, #48]
254.      ldp d12, d13, [sp, #32]
255.      ldp d10, d11, [sp, #16]
256.      ldp d8,  d9,  [sp], #64
257.      ret
```

7.3 gemv_opt function overview

The `gemv_opt` function uses the 4-way unsigned vertical dot product `uvdot` instruction.

The `uvdot` instruction is a multi-vector instruction:

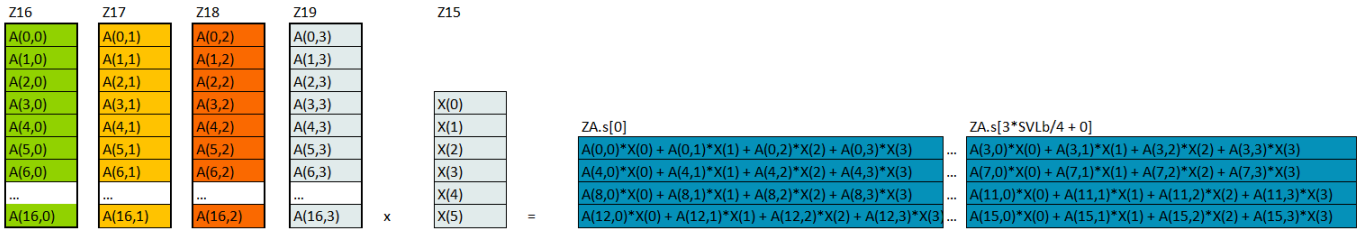
```
UVDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

`uvdot` takes four vectors in the first operand, `<Zn1>-<Zn4>`, and a single vector `<Zm>` in the second source.

The results are accumulated into a group of four ZA single-vectors. Each of these 4 vectors is then stored at the same index in its corresponding ZA array quarter. The vector index within a ZA quarter is determined by `Wv+offs`.

Figure 7-3: uvdot instruction register usage on page 91 shows how the calculation is performed using the `uvdot` instruction:

Figure 7-3: uvdot instruction register usage




```
uvdot za.s[row_base, 0, vgx4], { z16.b - z19.b }, z15.b[0]
```

The vectors where the results are stored are:

- `ZA.s[0]`
- `ZA.s[SVLb/4+0]`
- `ZA.s[2*SVLb/4+0]`
- `ZA.s[3*SVLb/4+0]`

The overall structure of the `gemv_opt` function is as follows:

```
_row_loop:
  // Iterate over the rows of the matrix (M dimension), in blocks
  // of SVLh x SVLh rows.

  col_loop:
    // Iterate over the columns of A, accumulating results for the current
    // set of rows in ZA tiles, processing in blocks of 16 columns at a time.

  _store_loop:
    // Move the results from ZA vectors to Z registers, de-interleave, and
    // store to memory.
```

Results are calculated as follows:

- The first four consecutive columns from matrix `A` are multiplied with the four first elements in `X`, performing a dot product calculation for each of the four multiplications.

  The register of the second operand (register Z15 in the example diagram above) has an immediate index between 0 and 3 which selects a 32-bit block for every 128-bit chunk. The 32-bit block holds four unsigned 8-bit integer elements. Because we loaded Z15 with `ld1rqb`, all 128-bit chunks in Z15 are the same. Because the index is also the same for all chunks, that means all of Z15 will have the same 32-bit block, containing the same four elements of X.
- The result of the `uvdot` dot product of four unsigned 8-bit integer elements is widened from unsigned 8-bit to unsigned 32-bit, indicated by the `.B` and `.S` suffixes. This means that the four input vectors still result in 4 output vectors.
- To calculate result vector B, we accumulate dot products for all the matrix column-by-vector element multiplications, accumulating over `a_cols`.
- The `uvdot` instruction produces deinterleaved data, as follows:

The first ZA vector accumulates the multiplications of rows of A by X, where the row index in A modulo 4 is 0. The second ZA vector accumulates the multiplications of rows of A by X, where the row index in A modulo 4 is 1, and so on. That is:

- `ZA.s[0]` accumulates the multiplications of rows of `A%4 = 0` by X
- `ZA.s[SVLb/4+0]` accumulates the multiplications of rows of `A%4 = 1` by X
- `ZA.s[2*SVLb/4+0]` accumulates the multiplications of rows of `A%4 = 2` by X
- `ZA.s[3*SVLb/4+0]` accumulates the multiplications of rows of `A%4 = 3` by X

Subsequent iterations repeat over all rows of A, in blocks of SVLh by SVLh rows.

This corresponds to the ZA calculations shown in blue in the above diagram.

The `gemv_opt` function uses the `uvdot` instruction because the data of A is arranged as column major. The four-way dot product multiplies four-by-four unsigned 8-bit integer elements and accumulates into a single unsigned 32-bit integer element.

However, this example requires that consecutive matrix elements in memory are loaded into a register. In column-major form, consecutive data values are the same column but different rows, while the dot product calculation requires 4 numbers from the same row. So instead, we load 4 registers consecutively for a single column (Z16, Z20, Z24, Z28), then we load 4 consecutive registers again but for the next column (z17, z21, z25, z29), and we do the same for the next two columns too. At this point the first element of registers z16-z19 are four consecutive elements of the same row. Therefore, the `uvdot` instruction is used here rather than the `udot` instruction, because the `uvdot` instruction calculates the dot product of four unsigned 8-bit integers from the same lane of the four first operand vectors by the 4 X values in the Z15 block that corresponds to their location.

7.4 gemv_opt function details

This section describes how the `gemv_opt` function operates, looking at sections of the code in turn.

- Line 176 (`gemv_opt`):

  The main `gemv_opt` function starts at line 176.
- Line 204 (`row_loop`):

  The code iterates over all rows. Each iteration of `row_loop` deals with a block of SVLh x SVLh rows. That is, `for i=0; i< M; i+=SVLh*SVLh`.
- Line 210:

  There are SVLb vectors in a ZA array, and each can hold SVLs results. Each result corresponds to one row of the input matrix. Each `col_loop` macro call processes SVLb x SVLs = SVLh x SVLh rows. The `csel` instruction conditionally selects the total number of ZA vectors to be used for accumulation before processing with the inner column loop, `col_loop`.
- Lines 213-214:

For each iteration, initialize the pointers to the memory positions of the data in `A` and `x`.

- Line 219:

  Call the `col_loop` macro, which loops over the number of columns, processing all columns of SVLh * SVLh rows.

- Line 47:

  The `ld1rqb` instruction loads 16 8-bit elements, or less if the predicate mask is not all true, from `x` to a 128-bit vector, then replicates that 128-bit vector over all 128 bits segments in Z15.

- Lines 56-160 (`_inner_row_loop`):

  The values loaded from `x` are multiplied by the elements of a matrix with `(16 x SVLb)` rows x 16 columns, or fewer if there are not enough rows remaining.

  `inner_row_loop` iterates over all ZA array vectors using `uvdot` instructions to multiply `A[16 x SVLb rows x 16 columns]` by `x[16 x 1]` and produce SVLh x SVLh results, before looping back and loading a new set of values from `x`.

- Lines 64-67:

  Two-dimensional predicate-as-counter for loads, built with the `psel` instruction.

  `row_pred` shows how many rows are left by having a bit set for every existing row in the next SVLb / 8 (bit per byte) rows. `row_pred` is filled with 1s until the final iteration, when the leftover rows are dealt with. For example, if only five rows remain then five bits are set in `row_pred`.

  The immediate index into `row_pred` in the `psel` instruction determines which bit is consulted. If that bit is 1, then `col_pred_p` is used as the predicate for that column. `col_pred_p` indicates how many columns are left. If the bit is 0, the predicate for that column is all false, which results in no loads or calculations for that out-of-bounds column.

  This 2D predication prevents overshoot for both columns and rows.

- Lines 70-73:

  Load four vectors from each of the four consecutive columns. The first `ld1b` instruction, at line 70, populates four Z vector registers with strided numbering, as follows:

  - Z16 = first SVLb bytes from the offset of column 0
  - Z20 = second SVLb bytes from the offset of column 0
  - Z24 = third SVLb bytes from the offset of column 0
  - Z28 = fourth SVLb bytes from the offset of column 0

  Because A is column-major, consecutively loading four register results in Z16, Z20, Z24, and Z28 holding consecutive elements of that single column.

  Similarly, the remaining `ld1b` instructions in lines 71-73 load Z vector registers as follows:

  - Z17, Z21, Z25, and Z29 from column 1

- Z18, Z22, Z26, and Z30 from column 2
- Z19, Z23, Z27, and Z31 from column 3

Using a multi-vector load instruction to a strided numbered group of Z vector registers means that Z registers are properly allocated for the `uvdot` instructions, with no need for additional `mov` instructions. The `uvdot` instruction requires four consecutive Z registers for its first operand, with the first register number being a multiple of 4. By contrast, the `ld1b` instruction does not require consecutive register numbers. Because the `uvdot` instruction requires a register from each of the 4 columns, the destination registers of the `ld1b` instruction are spaced four registers apart. After executing the four `ld1b` instructions, we can call `uvdot` using registers z16-19. And similarly for z20-z23, z24-z27, and z28-z31. By considering the destination registers used for `ld1b`, we prevented having to use `mov` instructions to move values into appropriately numbered registers for the `uvdot` instructions.

- Lines 79-82:

  Four `uvdot` instructions are used to compute the vertical dot products of the 4 x SVLb column elements loaded with the first four 8-bit elements from `x`, index 0 in `z15.b`.

  The vertical dot product is calculated by accumulating the 4-way dot-product to each result, where four contributing matrix elements are each from different first source operand vector register. That is, each element is from a different column. For example, the `uvdot` instruction at line 79 takes one element from each of z16, z17, z18, and z19.

  This process is replicated four times in the code that follows, loading the next four vectors from each of the next four consecutive columns and computing dot products using `uvdot` with incrementing index values. In this way, 16 matrix columns are loaded and multiplied by 16 8-bit X elements, because 128b/8b = 16. Therefore, `col_loop` performs a 16-way dot-product accumulate per result, because 16 columns and 16 x elements are consumed. This is because the `ld1rqb` instruction replicates 16 int8 elements, so unrolling the loop by a factor of four means that we fully use the replicated 16 int8s.

- Lines 225-240 (`_store_loop`)

  When all columns are processed for the SVLh x SVLh rows, the result is available in the ZA array. The final step is to store the data from the ZA array to memory.

  `_store_loop` does the following:

  - Line 227 moves a group of four ZA single-vectors to four consecutive Z vector registers using a four-vector `mov` instruction.
  - Line 231 interleaves the four extracted vectors using a four-vector `zip` instruction. The results were de-interleaved by the `uvdot` instruction, so now they are interleaved to recover the order. The `zip` instruction takes vectors z0-z3 and interleaves them so that z0 becomes `z3[1],z2[1],z1[1],z0[1] z3[0],z2[0],z1[0],z0[0]`.
  - Line 234 performs a contiguous four-vector `st1w` store to the current memory address for result vector `B`.

8. lut_gemv_rm_int8: Compressed 8-bit integer to 32-bit integer matrix-by-vector multiplication

The lut_gemv_rm_int8 example implements generalized matrix-by-vector multiplication (gemv) of a compressed matrix `A_compressed` and an uncompressed vector, where values in the matrix are compressed, or quantized, from 8 bits to 2 bits per matrix element.

8.1 Overview of the lut_gemv_rm_int8 algorithm

The lut_gemv_rm_int8 example shows how to use `luti2` lookup table instructions to decompress compressed elements in a matrix, and then perform a general row-major matrix-by-vector multiplication using those decompressed values. Decompression is performed on-the-fly, in the body of the inner-most loop, `_inner_row_loop`.

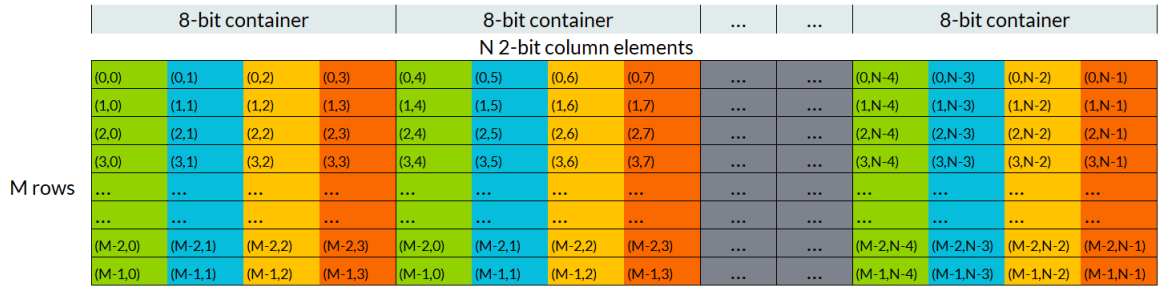
Each value in the matrix is compressed from 8 bits to 2 bits per matrix element. This means that only four uint8 values are possible:

- `00` decompresses to `00000000`
- `01` decompresses to `01000000`
- `10` decompresses to `10000000`
- `11` decompresses to `11000000`

The compressed matrix is stored in a row-major format in memory.

Figure 8-1: Example 2-bit compressed matrix memory layout on page 95 shows an example memory layout with 4 values per byte. Each row is a combination of four compressed 2-bit values stored as a single byte in memory:

Figure 8-1: Example 2-bit compressed matrix memory layout



8.2 lut_gemv_opt code

The following code shows the `lut_gemv_opt` function with numbered lines. Subsequent sections explain how the code operates.

```
1.  // Input parameters
2.  #define A         x0 // Row-major matrix of compressed data (const uint8_t *)
3.  #define x         x1 // Vector 'x' (const uint8_t *)
4.  #define B         x2 // Output vector 'B = Ax' (uint32_t *B)
5.  #define lut       x3 // Lookup table (const uint8_t *)
6.  #define a_rows    x4 // Number of rows in 'A' (uint64_t)
7.  #define a_cols    x5 // Number of cols in 'A' (uint64_t)
8.
9.  // Working registers
10. #define A_row_ptr           x6  // Pointer to iterate through rows of 'A'
11. #define A_col_ptr           x7  // Pointer to iterate within rows of 'A'
12. #define A_end_ptr           x8  // End of the input matrix 'A'
13. #define x_vec_ix            x9  // Loop index for columns of A
14. #define row_base            w10 // Loop index for row group (ZA vector select)
15. #define row_base_odd_vecs   x11 // Index of the second of the four vectors in
16.                                    // a row group (= row_base + cntw)
17. #define row_base_odd_vecs_w w11 // (Above as 32-bit register)
18. #define rows_remaining      x12 // Number of rows remaining to process
19. #define slice_row_base      w13 // Copy of `row_base` in range w12-w15
20. #define rows_to_process     x14 // Number of rows to process in this iteration
21. #define rows_to_process_w   w14 // (Above as 32-bit register)
22. #define tmp0                w15 // Temporary register
23. #define a_row_stride        x16 // 'A' row stride
24. #define a_row_stride_2x     x17 // 'A' two row stride
25. #define a_row_stride_3x     x18 // 'A' three row stride
26. #define a_row_stride_4x     x19 // 'A' four row stride
27. #define a_col_ix            x20 // Loop index for columns of A
28.
29. // Predicates
30. #define all_bytes    p0   // Byte predicate with mask: 111111111...
31. #define four_words   p1   // Word predicate with mask: 100100001000100000...
32. #define col_pred     p2   // Byte predicate for the M dimension
33. #define store_pred   p3   // Byte predicate used for storing results
34. #define x_pred_n     pn8  // Predicate-as-counter for X loads
35. #define row_pred_p   p9   // Predicate-as-mask for row loads
36. #define row_1_pred_p p4   // (Above in predicate-as-mask form)
37. #define row_2_pred_p p5   // (Above in predicate-as-mask form)
38. #define row_3_pred_p p6   // (Above in predicate-as-mask form)
39. #define row_4_pred_p p7   // (Above in predicate-as-mask form)
40.
41. // void lut_gemv_opt(const uint8_t *A, const uint8_t *x, uint32_t *B,
42. //             const uint8_t *lut, uint64_t a_rows, uint64_t a_cols);
43. .text
44. .align 2
45. .global lut_gemv_opt
46. .type   lut_gemv_opt, %function
47. lut_gemv_opt:
48.   // Spill callee-saved registers
49.   stp x19, x20, [sp, #-16]!
50.
51.   // Enter streaming mode
52.   smstart
53.
54.   // Load LUT
55.   ldr zt0, [lut]
56.
57.   // Initialise constants
58.   mov A_row_ptr, A                      // A_row_ptr = A
59.   mov A_col_ptr, A                      // A_col_ptr = A
60.   ptrue all_bytes.b
61.   ptrue four_words.s, VL4
62.   cntw rows_to_process                  // rows_to_process = SVLs
```

```
63.     mov rows_remaining, a_rows              // rows_remaining = a_rows
64.     add a_row_stride, a_cols, #3
65.     lsr a_row_stride, a_row_stride, #2      // a_row_stride = (a_cols+3) >> 2
66.     lsl a_row_stride_2x, a_row_stride, #1   // 2*a_row_stride
67.     add a_row_stride_3x, a_row_stride, a_row_stride, lsl #1 // 3*a_row_stride
68.     lsl a_row_stride_4x, a_row_stride, #2   // 4*a_row_stride
69.     madd A_end_ptr, a_rows, a_row_stride, A // A_end_ptr=A+a_rows*a_row_stride
70.
71.     // Row loop: iterate through rows of A in chunks of `rows_to_process`,
72.     // processing a minimum of four rows (and producing a minimum of four
73.     // 8-bit result elements in 'B') at each iteration
74.     cmp A_row_ptr, A_end_ptr
75.     b.gt ._row_loop_end
76.     ._row_loop:
77.       // Zero all tiles (prepare to accumulate new results)
78.       zero {za}
79.
80.       // Calculate the number of rows to process this iteration
81.       cmp rows_to_process, rows_remaining
82.       csel rows_to_process, rows_to_process, rows_remaining, lt
83.       whilelt col_pred.b, xzr, rows_remaining
84.
85.       // Column loop: iterate through the columns of A, accumulating partial
86.       // products from `rows_to_process` 8-bit result elements in ZA,
87.       // processing in chunks of 4*SVL columns at a time
88.       mov x_vec_ix, #0                              // x_vec_ix = 0
89.       whilelt x_pred_n.b, x_vec_ix, a_cols, vlx4    // Prepare 4*SVL X pred
90.
91.       mov a_col_ix, #0                              // a_col_ix = 0
92.       whilelt row_pred_p.b, a_col_ix, a_row_stride // Prepare row pred
93.
94.       ._col_loop:
95.         // Load 4*SVL 8-bit elements from 'x'
96.         ld1b { z28.b - z31.b }, x_pred_n/z, [x, x_vec_ix]
97.
98.         // Inner row loop: iterate through `rows_to_process` in groups
99.         // of four rows, loading data from the 4*SVL columns associated
100.        // with the 'x' data in z28-z31 and using UDOT to accumulate
101.        // results into ZA
102.        mov row_base, #0                    // row_base = 0
103.        add A_col_ptr, A_row_ptr, a_col_ix // A_col_ptr = A_row_ptr + a_col_ix
104.        ._inner_row_loop:
105.          // Combine row and column predicates to avoid loading beyond
106.          // the end of the matrix (no awkward tail loops required!)
107.          mov slice_row_base, row_base
108.          psel row_1_pred_p, row_pred_p, col_pred.b[slice_row_base, 0]
109.          psel row_2_pred_p, row_pred_p, col_pred.b[slice_row_base, 1]
110.          psel row_3_pred_p, row_pred_p, col_pred.b[slice_row_base, 2]
111.          psel row_4_pred_p, row_pred_p, col_pred.b[slice_row_base, 3]
112.
113.          // Load 4*SVLb uint2 elements from each of four columns in 'A'
114.          ld1b { z24.b }, row_1_pred_p/z, [A_col_ptr]
115.          ld1b { z25.b }, row_2_pred_p/z, [A_col_ptr, a_row_stride]
116.          ld1b { z26.b }, row_3_pred_p/z, [A_col_ptr, a_row_stride_2x]
117.          ld1b { z27.b }, row_4_pred_p/z, [A_col_ptr, a_row_stride_3x]
118.
119.          // Expand uint2 elements to uint8
120.          luti2 {  z0.b -  z3.b }, zt0, z24[0]
121.          luti2 {  z4.b -  z7.b }, zt0, z25[0]
122.          luti2 { z16.b - z19.b }, zt0, z26[0]
123.          luti2 { z20.b - z23.b }, zt0, z27[0]
124.
125.          // Take the dot product between groups of four elements in the
126.          // loaded rows of 'A' and elements of 'x', calculating
127.          // partially reduced results for each result element in 'B'
128.          // (accumulating widened 8-bit -> 32-bit results in ZA).
129.          udot za.s[row_base, 0, vgx4], {  z0.b -  z3.b }, { z28.b - z31.b }
130.          udot za.s[row_base, 1, vgx4], {  z4.b -  z7.b }, { z28.b - z31.b }
131.          udot za.s[row_base, 2, vgx4], { z16.b - z19.b }, { z28.b - z31.b }
132.          udot za.s[row_base, 3, vgx4], { z20.b - z23.b }, { z28.b - z31.b }
133.
```

```
134.         // Update inner row loop variables
135.         add row_base, row_base, #4                // row_base  += 4 rows
136.         add A_col_ptr, A_col_ptr, a_row_stride_4x // A_col_ptr += 4 rows
137.         cmp row_base, rows_to_process_w
138.         b.lt ._inner_row_loop
139.
140.       // Update column loop variables
141.       addvl x_vec_ix, x_vec_ix, #4                 // x_vec_ix += 4*SVLb
142.       whilelt x_pred_n.b, x_vec_ix, a_cols, vlx4   // Prepare 4*SVL X pred
143.
144.       addvl a_col_ix, a_col_ix, #1                 // a_col_ix += SVLb
145.       whilelt row_pred_p.b, a_col_ix, a_row_stride // Prepare row pred
146.
147.       b.first ._col_loop
148.
149.     // Store loop: iterate through `rows_to_process` in groups of four
150.     // rows, reducing partial products and storing results, writing out
151.     // up to four result elements in 'B' at each iteration
152.     mov row_base, #0                             // row_base = 0
153.     cntw row_base_odd_vecs                       // row_base_odd_vecs = SVLs
154.     whilelt store_pred.s, wzr, rows_to_process_w // Prepare store pred
155.     ._store_loop:
156.       // Prepare predicate for storing up to four elements of 'B'
157.       mov store_pred.b, four_words/z, store_pred.b
158.
159.       // Move 'odd vector' results (vectors one and three out of the four
160.       // per row group) out of ZA and into Z
161.       mova {  z2.d -  z3.d }, za.d[row_base_odd_vecs_w, 0, vgx2]
162.       mova {  z6.d -  z7.d }, za.d[row_base_odd_vecs_w, 1, vgx2]
163.       mova { z18.d - z19.d }, za.d[row_base_odd_vecs_w, 2, vgx2]
164.       mova { z22.d - z23.d }, za.d[row_base_odd_vecs_w, 3, vgx2]
165.
166.       // Add 'odd vectors' to 'even vectors' to partially reduce
167.       add za.s[row_base, 0, vgx2], {  z2.s -  z3.s }
168.       add za.s[row_base, 1, vgx2], {  z6.s -  z7.s }
169.       add za.s[row_base, 2, vgx2], { z18.s - z19.s }
170.       add za.s[row_base, 3, vgx2], { z22.s - z23.s }
171.
172.       // Reduce 'even vectors'
173.       mova {  z0.d -  z1.d }, za.d[row_base, 0, vgx2]
174.       mova {  z4.d -  z5.d }, za.d[row_base, 1, vgx2]
175.       mova { z16.d - z17.d }, za.d[row_base, 2, vgx2]
176.       mova { z20.d - z21.d }, za.d[row_base, 3, vgx2]
177.       add  z0.s,  z0.s,  z1.s
178.       add  z4.s,  z4.s,  z5.s
179.       add z16.s, z16.s, z17.s
180.       add z20.s, z20.s, z21.s
181.       uaddv d0, all_bytes,  z0.s
182.       uaddv d1, all_bytes,  z4.s
183.       uaddv d2, all_bytes, z16.s
184.       uaddv d3, all_bytes, z20.s
185.
186.       // Pack results into the first four elements of z0.s
187.       zip { z0.s - z3.s }, { z0.s - z3.s }
188.
189.       // Store
190.       st1w z0.s, store_pred, [B]
191.
192.       // Update store loop variables
193.       add B, B, #16                                // B += 4 elements
194.       add row_base, row_base, #4                   // row_base+=4 rows
195.       add row_base_odd_vecs, row_base_odd_vecs, #4   // row_base_odd_vecs+=4
196.       whilelt store_pred.s, row_base, rows_to_process_w // Prepare store pred
197.       b.first ._store_loop
198.
199.     // Update outer row loop variables
200.     madd A_row_ptr, a_row_stride, rows_to_process, A_row_ptr
                                    // A_row_ptr += `rows_to_process` rows
201.     sub rows_remaining, rows_remaining, rows_to_process
                                    // rows_remaining -= `rows_to_process`
202.     cmp A_row_ptr, A_end_ptr
```

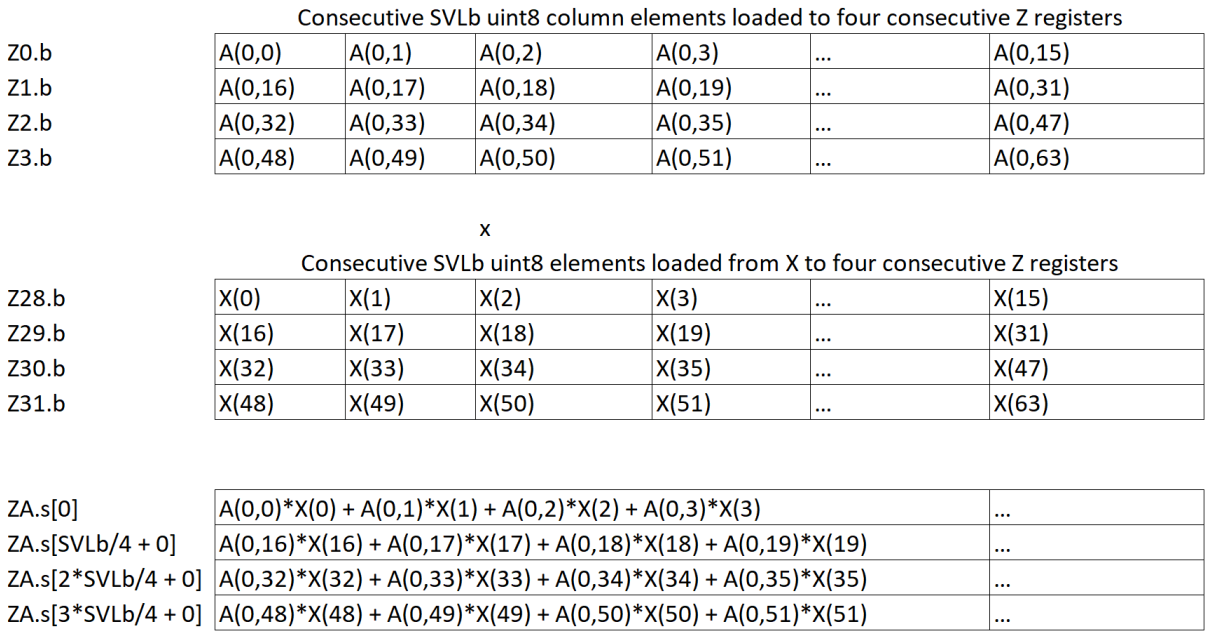
```
203.      b.lt ._row_loop
204.    ._row_loop_end:
205.
206.    // Exit streaming mode and return
207.    smstop
208.
209.    // Restore callee-saved registers and return
210.    ldp x19, x20, [sp], #16
211.
212.    ret
```

8.3 lut_gemv_opt function overview

The `lut_gemv_opt` function performs multiplication using a four-way dot-product-and-accumulate computation with the four-vector `udot` instruction.

Figure 8-2: Four-way dot-product and accumulate computation on page 99 shows the process where SVLb is 16 and therefore SVLs is 4.

Figure 8-2: Four-way dot-product and accumulate computation



The function does the following:

- Loads one vector of compressed data from a single row of matrix `A` and decompresses it to four vectors
- Loads four consecutive vectors from vector `x`
- Performs the dot-product of row `A` and vector `x`.
- Accumulates the result in four vectors, one in each ZA array quarter.

8.4 lut_gemv_opt function details

This section describes how the `lut_gemv_opt` function operates, looking at sections of the code in turn.

- Line 47 (`lut_gemv_opt`)

  `lut_gemv_opt` consumes a maximum of SVLb/4 rows per `_row_loop` iteration. The ZA array has SVLb vectors available, and each group of four ZA single-vectors is used to perform the dot product of four vectors from a single compressed row from `A`.

  The function iterates over all rows (`._row_loop` in line 76) and columns (`.col_loop` in line 94).
- Line 55:

  The `ldr` instruction loads the compressed data from memory to the lookup table register ZT0.

  ZT0 is a dedicated 512-bit lookup table register for data decompression.
- Line 96:

  The `ld1b` multi-vector instruction loads four consecutive vectors from X.
- Lines 108 to 111:

  The `psel` instructions perform 2D predication to manage the vertical and horizontal matrix edges for compressed matrix one-vector loads.
- Lines 114 to 123:

  Each `ld1b` instruction loads a single vector from the compressed A matrix. Four rows are loaded to four consecutive Z registers (Z24 to Z27). Each vector register is populated with compressed 2-bit row elements.

  Each `luti2` instruction decompresses matrix elements by looking up the decompressed value from ZT0 using the 2-bit compressed elements in Z to obtain 8-bit inputs. Four Z vectors are created from each compressed Z vector.

  Index `[0]` in `z24[0]` indicates that the instruction uses the segment of z24 from bit 0. SVLs (SVLb/4) 2-bit indices from z24 bits `[0:SVL/4-1]` are decompressed to z0, then the next SVLs indices are decompressed to z1, and so on. The 2-bit indices address the first four entries of the lookup table in ZT0.
- Lines 129-132

  The four-vector `udot` instructions perform 4-way dot product accumulation per destination element.

  After the `udot` instructions, the following ZA array vectors contain 32-bit integer partial sums of the B result:

  - `ZA.s[idx]`
  - `ZA.s[SVLb/4+idx]`

    - `ZA.s[2*SVLb/4+idx]`
    - `ZA.s[3*SVLb/4+idx]`

- Lines 155 to 197 (`_store_loop`):

  The `udot` instructions at lines 129-132 produce partial sums of products in 4 ZA single-vectors. To obtain the final inner product result, we need to sum these partial sums of products. The `_store_loop` function performs the accumulation of partial sums, that is the sum of all `A(I,:) * X(:)` multiplications.

  For each of the four ZA single partial product (that is, the result of processing a single `A` row):

  - Extract the second group of 2 vectors and add them to the first group of two vectors using the multi-vector add instructions (lines 161 to 170). Reduce from four to two vectors.
  - Extract the result of the above addition, and perform a single vector add to reduce the result into a single vector (lines 173 to 180).
  - Reduce the above result into a single 32-bit element using `uaddv` instructions. (lines 181 to 184).

- Lines 187 and 190:

  Using the four-vector `zip` instruction, 32-bit results from consecutive rows are interleaved from consecutive rows into the z0 vector register. A single vector store instruction stores four 32-bit elements.

- Line 201:

  The `_row_loop` outer loop is incremented by SVLb/4.

9. cplx_matmul_fp16fp32: Complex-valued half-precision to single precision floating-point matrix-by-matrix multiplication

The cplx_matmul_fp16fp32 example implements complex-valued matrix-by-matrix multiplication with half-precision floating-point complex-valued inputs and outputs.

9.1 Overview of the cplx_matmul_fp16fp32 algorithm

The cplx_matmul_fp16fp32 example operates on the following matrices:

- `matLeft` is an M x K LHS input matrix containing IQ interleaved half-precision floating-point values.
- `matRight` is an K x N RHS input matrix containing IQ interleaved half-precision floating-point values.
- `matResult_opt` is an M x N containing the result of multiplying `matLeft` with `matRight` with IQ interleaved half-precision floating-point values.

The cplx_matmul_fp16fp32 example uses the sum of two outer-product widening `fmopa` instructions, which accumulates half-precision floating-point two-way dot-products into single-precision results in a 32-bit element ZA tile. The result is then down-converted to 16-bit half-precision floating-point.

Consider the complex-valued matrix-by-matrix multiplication `C = A x B`.

The first accumulated results are as follows:

```
ZA0.s[0] = Re[ A(0,0)*B(0,0) + A(0,1)*B(1,0) ]
ZA1.s[0] = Im[ A(0,0)*B(0,0) + A(0,1)*B(1,0) ]
```

Two ZA tiles are required to compute the results of the complex-valued multiplications:

- one to accumulate the real results
- one to accumulate the imaginary results

In this example, even-numbered ZA tiles accumulate the real results, and odd-numbered ZA tiles accumulate the imaginary results.

9.2 preprocess_l code

The following code shows the `preprocess_l` function with numbered lines. Subsequent sections explain how the code operates.

```
1.  preprocess_l:
2.    // cplx_mat_trans_opt(M, K, matLeft, matLeft_mod);
3.    // x0 : M
4.    // x1 : K, lda
5.    // x2 : matLeft
6.    // x3 : matLeft_mod
7.    // x4 : SVLs
8.    // x5 : SVLs*SVLs
9.    // x6 : a_ptr
10.   // x7 : store_offset0
11.   // x8 : store_offset1
12.   // x9 :
13.   // x10: Loop_M exit condition
14.   // x11: SVLs*lda
15.   // x12: Loop_load/Loop_store loop counter
16.   // x13: K loop exit condition
17.   // x14: mat_base
18.   // x15: c_ptr
19.
20. // Assumptions:
21. // nbr in matLeft (M): any
22. // nbc in matLeft, nbr in matRight (K): any K > 2
23. // nbc in matRight (N): any
24. //
25. // Left matrix rearrangement:
26. // The entire matrix is transposed by blocks of SVLs rows and contiguously
                                              stored in the memory.
27. // Output buffer is a multiple of SVLs cplx-FP16 elements
                                              (padded by zeros if applicable)
28.
29.   smstart
30.
31. // constants
32.   cntw    x4                      // SVLs
33.   mul     x5, x4, x4              // SVLs*SVLs
34.   mul     x11, x4, x1             // SVLs*lda
35.   mov     x10, #0                 // Loop_M counter
36.   whilelt p10.s, xzr, x0          // Load tile predicate (M dimension)
37.
38. .Loop_M:
39.   mov     x15, x3                 // c_ptr = &matLeft_mod
40.   mov     x14, x2                 // mat_base = &matLeft
41.   add     x13, x2, x1, lsl #2     // Loop_K exit condition
42.
43.   whilelt pn12.b, x14, x13, vlx2  // K dimension predicate-as-counter
44.
45.   mov     x7, #0                  // store_offset0
46.   mov     x8, x5                  // store_offset1
47.
48. .Loop_K:
49.   mov     x6, x14                 // a_ptr = mat_base
50.   mov     w12, #0                 // Loop_load counter
51. .Loop_load:
52.   psel     pn8, pn12, p10.s[w12, #0]
53.   psel     pn9, pn12, p10.s[w12, #1]
54.   ld1w    {z0.s,z8.s}, pn8/z, [x6]              // Load 1st row from mat_ptr
55.   ld1w    {z1.s,z9.s}, pn9/z, [x6, x1, lsl #2]  // Load 2nd row from mat_ptr
56.   mova    za0h.s[w12, 0:1], { z0.s-z1.s }
57.   mova    za1h.s[w12, 0:1], { z8.s-z9.s }
58.
59.   add     w12, w12, #2                // Loop_load counter increment
60.   add     x6, x6, x1, lsl #3          // a_ptr += 2*lda Cplx-FP16 elems
```

```
61.   cmp     w12, w4
62.   b.mi    .Loop_load
63.
64.   mov     w12, #0                     // Loop_store counter
65. .Loop_store:
66.   whilelt pn8.s, x7, x11, vlx4        // Tile0 store predicate-as-counter
67.   whilelt pn9.s, x8, x11, vlx4        // Tile1 store predicate-as-counter
68.   mova    {z0.s-z3.s}, za0v.s[w12, 0:3]
69.   mova    {z4.s-z7.s}, za1v.s[w12, 0:3]
70.   st1w    {z0.s-z3.s}, pn8, [x15, x7, lsl #2] // Store 4 cols of Tile0 to
                                                      matLeft_mod +
 store_offset0
71.   st1w    {z4.s-z7.s}, pn9, [x15, x8, lsl #2] // Store 4 cols of Tile1 to
                                                      matLeft_mod +
 store_offset1
72.   addvl   x7, x7, #1                  // Tile0 store pointer offset += 4*SVLs
73.   addvl   x8, x8, #1                  // Tile1 store pointer offset += 4*SVLs
74.   add     w12, w12, #4                // Loop_store counter increment
75.   cmp     w12, w4
76.   b.mi    .Loop_store
77.
78.   add     x7, x7, x5              // Tile0 store pointer offset += SVLs*SVLs
79.   add     x8, x8, x5              // Tile1 store pointer offset += SVLs*SVLs
80.   addvl   x14, x14, #2            // mat_base += 2*SVLs Cplx-FP16 elems
81.   whilelt pn12.b, x14, x13, vlx2  // K dimension predicate-as-counter
82.   b.first .Loop_K
83.
84.   add     x2, x2, x11, lsl #2     // &matLeft += SVLs*lda Cplx-FP16 elems
85.   incw    x10                     // Loop_M counter increment
86.   add     x3, x3, x11, lsl #2     // &matLeft_mod += SVLs*lda Cplx-FP16 elems
87.   whilelt p10.s, x10, x0          // M dimension predicate
88.   b.first .Loop_M
89.
90.   smstop
91.
92.   ret
```

9.3 preprocess_l function overview

The `preprocess_l` function re-arranges the `matLeft` matrix such that blocks of SVLs (rows) x K (columns) are transposed and contiguously stored to memory, in a similar way as the same function in the matmul_fp32 example.

Each 32-bit transposed element is the combination of the real and imaginary part of a single matrix element. Data is loaded as follows: `{Re(0), Im(0), Re(1), Im(1), ...)`. The input matrix is zero-padded to a multiple of SVLs rows.

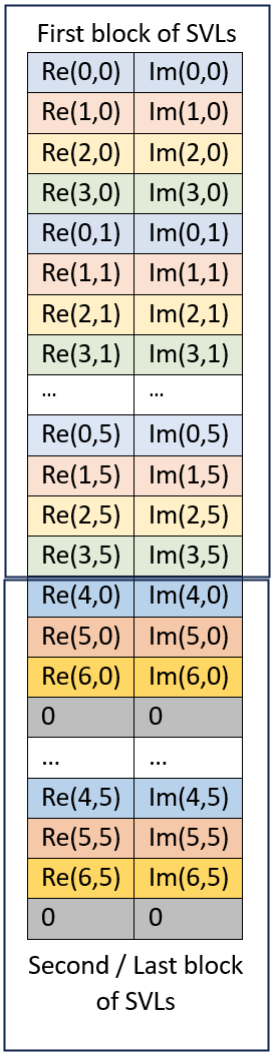
For example, consider the complex-valued half-precision floating-point `matLeft` matrix with 7 rows and 6 columns shown in Figure 9-1: Example matLeft matrix on page 105:

Figure 9-1: Example matLeft matrix

Re(0,0)	Im(0,0)	Re(0,1)	Im(0,1)	Re(0,2)	Im(0,2)	Re(0,3)	Im(0,3)	Re(0,4)	Im(0,4)	Re(0,5)	Im(0,5)
Re(1,0)	Im(1,0)	Re(1,1)	Im(1,1)	Re(1,2)	Im(1,2)	Re(1,3)	Im(1,3)	Re(1,4)	Im(1,4)	Re(1,5)	Im(1,5)
Re(2,0)	Im(2,0)	Re(2,1)	Im(2,1)	Re(2,2)	Im(2,2)	Re(2,3)	Im(2,3)	Re(2,4)	Im(2,4)	Re(2,5)	Im(2,5)
Re(3,0)	Im(3,0)	Re(3,1)	Im(3,1)	Re(3,2)	Im(3,2)	Re(3,3)	Im(3,3)	Re(3,4)	Im(3,4)	Re(3,5)	Im(3,5)
Re(4,0)	Im(4,0)	Re(4,1)	Im(4,1)	Re(4,2)	Im(4,2)	Re(4,3)	Im(4,3)	Re(4,4)	Im(4,4)	Re(4,5)	Im(4,5)
Re(5,0)	Im(5,0)	Re(5,1)	Im(5,1)	Re(5,2)	Im(5,2)	Re(5,3)	Im(5,3)	Re(5,4)	Im(5,4)	Re(5,5)	Im(5,5)
Re(6,0)	Im(6,0)	Re(6,1)	Im(6,1)	Re(6,2)	Im(6,2)	Re(6,3)	Im(6,3)	Re(6,4)	Im(6,4)	Re(6,5)	Im(6,5)

After processing by the `preprocess_l` function, `matLeft_mod` contains the processed matrix. With SVL=128b, blocks of 4 x 6 columns are transposed and contiguously stored to the memory. Figure 9-2: matLeft_mod memory layout on page 105 shows the memory layout of `matLeft_mod`, with each row occupying 4 bytes:

Figure 9-2: matLeft_mod memory layout



9.4 preprocess_l function details

This section describes how the `preprocess_l` function operates, looking at sections of the code in turn.

- Lines 54-57 (`Loop_load`):

  In lines 54 and 55, the two-vector `ld1w` instruction loads contiguous elements from one row into two vector registers with strided numbering.

  In lines 56 and 57, the two-vector `mova` instruction moves elements of two matrix rows from consecutively numbered Z vector registers to consecutive horizontal slices of one 32-bit ZA tile.

  These instructions, together with the instructions at lines 68-71, perform 32-bit width transposition. 32-bit element granularity is used because each input occupies 32 bits: 16 bits for the real part, and 16 bits for the imaginary part.

  This transposition is very similar to the same function in the matmul_fp32 example.

- Lines 68-71 (`Loop_store`)

  The `mova` instructions extract four consecutive vertical slices from 32-bit element ZA tiles to four Z vector registers. The four-vector `st1w` instructions then store these vectors to consecutive memory locations.

9.5 cplx_matmul_opt code

The following code shows the `cplx_matmul_opt` function with numbered lines. Subsequent sections explain how the code operates.

```
1.   cplx_matmul_opt:
2.     // cplx_matmul_opt(M, K, N, matLeft, matRight, matResult_opt);
3.     // x0 : M, lda
4.     // x1 : K
5.     // x2 : N, ldb, ldc
6.     // x3 : matLeft
7.     // x4 : matRight
8.     // x5 : matResult_opt
9.     // x6 : SVLs*ldc
10.    // x7 : SVLb - 4
11.    // x8 : a_ptr
12.    // x9 : Loop_M counter
13.    // x10: K*ldb
14.    // x11: matRight end address
15.    // x12: c_base
16.    // x13: Loop_N exit condition
17.    // x14: Loop_store counter
18.    // x15: Loop_K exit condition
19.    // x16: c_ptr
20.    // x17: b_ptr
21.    // x18: K*SVLb
22.    // x19: b_base
```

```
23.
24.  // Assumptions:
25.  // nbr in matLeft (M): any
26.  // nbc in matLeft, nbr in matRight (K): any K > 2
27.  // nbc in matRight (N): any
28.  //
29.  // Left matrix is pre-arranged.
30.  //
31.  // 32-bit accumulator mapping with 2x2 tiles processing
32.
33.    str     x19, [sp, #-16]!
34.
35.    smstart
36.
37.  // constants
38.    cntb    x7                  // SVLb
39.    mul     x10, x1, x2         // K*ldb
40.    mul     x6, x7, x2          // SVLs*ldc Cplx-FP16 elems
41.
42.    mul     x18, x1, x7         // K*SVLb
43.    sub     x7, x7, #4          // SVLb - 4
44.
45.    ptrue   p5.b
46.    pfalse  p6.b
47.    trn1    p6.h, p6.h, p5.h    // set even elems true, odd elems false
48.
49.    mov     x9, #0              // Loop_M counter
50.    whilelt p2.s, x9, x0        // M dimension predicate
51.    ptrue   pn8.b               // Predicate as counter for SME2 VLx2 LD1H
                                                              (a_ptr loads)
52.    add     x13, x4, x2, lsl #2  // Loop_N exit condition
53.
54. .Loop_M:
55.    mov     x19, x4             // b_base = &matRight
56.    mov     x12, x5             // c_base = &matResult_opt
57.
58.    whilelt pn9.b, x19, x13, vlx2  // N dimension predicate-as-counter
59.
60. .Loop_N:
61.    mov     x8, x3              // a_ptr = &matLeft
62.    mov     x17, x19            // b_ptr = b_base
63.    mov     x16, x12            // c_ptr = c_base
64.
65.    pext    { p0.b, p1.b }, pn9[0] // Tile0/2 predicates
66.
67.    zero    {za}
68.    ld1w    {z9.s}, p5/z, [x8]         // Load 1st vector from a_ptr
69.    addvl   x8, x8, #1                 // a_ptr += SVLb
70.
71.    revh    z1.s, p5/m, z9.s           // z1  = {Im, Re, Im, Re, ...}
72.    fneg    z9.h, p6/m, z9.h           // z9  = {Re,-Im, Re,-Im, ...}
73.    add     x11, x17, x10, lsl #2      // matRight end address
74.
75.    ld1w    {z2.s-z3.s}, pn9/z, [x17]  // Load 2 vectors from b_ptr
76.    sub     x15, x11, x2, lsl #2       // Loop_K exit condition
77.    add     x17, x17, x2, lsl #2       // b_ptr += ldb Cplx-FP16 elems
78.
79. .Loop_K:
80.    fmopa   za0.s, p5/m, p0/m, z9.h, z2.h  // ZA0 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                      1st part
81.
82.    fmopa   za2.s, p5/m, p1/m, z9.h, z3.h  // ZA2 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                      2nd part
83.    ld1w    {z8.s-z9.s}, pn8/z, [x8]       // Load next 2 vectors from a_ptr
84.
85.    fmopa   za1.s, p5/m, p0/m, z1.h, z2.h  // ZA1 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                      1st part
86.    ld1w    {z6.s-z7.s}, pn9/z, [x17]      // Load next 2 vectors from b_ptr
87.    revh    z5.s, p5/m, z8.s               // z5  = {Im, Re, Im, Re, ...}
88.
89.    fmopa   za3.s, p5/m, p1/m, z1.h, z3.h  // ZA3 += {ImRe+ReIm, ImRe+ReIm, ...}
```

```
                                                                    2nd part
90.    fneg    z8.h, p6/m, z8.h               // z8  = {Re,-Im, Re, -Im, ...}
91.
92.    fmopa   za0.s, p5/m, p0/m, z8.h, z6.h  // ZA0 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                    1st part
93.    addvl   x8, x8, #2                     // a_ptr += 2*SVLb
94.
95.    fmopa   za2.s, p5/m, p1/m, z8.h, z7.h  // ZA2 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                    2nd part
96.
97.    fmopa   za1.s, p5/m, p0/m, z5.h, z6.h  // ZA1 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                    1st part
98.    revh    z1.s, p5/m, z9.s               // z1  = {Im, Re, Im, Re, ...}
99.
100.   fmopa   za3.s, p5/m, p1/m, z5.h, z7.h  // ZA3 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                    2nd part
101.   ld1w    {z2.s-z3.s}, pn9/z, [x17, x2, lsl #2] // Load 2 vecs from b_ptr+ldb
102.   fneg    z9.h, p6/m, z9.h                 // z9  = {Re,-Im, Re,-Im, ...}
103.   add     x17, x17, x2, lsl #3             // b_ptr += 2*ldb cplx-FP16 elems
104.
105.   cmp     x17, x15
106.   b.lt    .Loop_K
107.
108.   fmopa   za0.s, p5/m, p0/m, z9.h, z2.h  // ZA0 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                    1st part
109.
110.   fmopa   za2.s, p5/m, p1/m, z9.h, z3.h  // ZA2 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                    2nd part
111.
112.   fmopa   za1.s, p5/m, p0/m, z1.h, z2.h  // ZA1 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                    1st part
113.
114.   fmopa   za3.s, p5/m, p1/m, z1.h, z3.h  // ZA3 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                    2nd part
115.
116.   cmp     x17, x11
117.   b.ge    .Ktail_end
118.
119. .Ktail_start:
120.   ld1w    {z0.s}, p5/z, [x8]             // Load 1 vector from a_ptr
121.   ld1w    {z2.s-z3.s}, pn9/z, [x17]      // Load 2 vectors from b_ptr
122.   revh    z1.s, p5/m, z0.s               // z1  = {Im, Re, Im, Re, ...}
123.   fneg    z0.h, p6/m, z0.h               // z0  = {Re, -Im, Re, -Im, ...}
124.
125.   fmopa   za0.s, p5/m, p0/m, z0.h, z2.h  // ZA0 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                    1st part
126.
127.   fmopa   za2.s, p5/m, p1/m, z0.h, z3.h  // ZA2 += {ReRe-ImIm, ReRe-ImIm, ...}
                                                                    2nd part
128.
129.   fmopa   za1.s, p5/m, p0/m, z1.h, z2.h  // ZA1 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                    1st part
130.
131.   fmopa   za3.s, p5/m, p1/m, z1.h, z3.h  // ZA3 += {ImRe+ReIm, ImRe+ReIm, ...}
                                                                    2nd part
132.
133. .Ktail_end:
134.   mov     w14, #0                        // Loop_store counter
135.
136. // Prologue
137.   // Z0=ZA0H.S[0], Z1=ZA1H.S[0], Z2=ZA2H.S[0] and Z3=ZA3H.S[0]
138.   // -- Z0, Z2 = {ReRe-ImIm; ReRe-ImIm; ...}
139.   // -- Z1, Z3 = {ReIm+ImRe; ReIm+ImRe; ...}
140.   mova    {z0.b-z3.b}, za0h.b[w14, 0:3]
141.
142. .Loop_store:
143.   fcvtn   z8.h, {z0.s-z1.s}              // Convert and interleave
144.   fcvtn   z9.h, {z2.s-z3.s}              // Convert and interleave
145.   psel    pn10, pn9, p2.b[w14, 0]
146.   st1w    { z8.s-z9.s }, pn10, [x16]     // Store 2 vectors to c_ptr
147.   add     x16, x16, x2, lsl #2           // c_ptr += ldc Cplx-FP16 elems
```

```
148.    add     w14, w14, #4                    // Loop_store counter increment
149.
150.    // Z0=ZA0H.S[n], Z1=ZA1H.S[n], Z2=ZA2H.S[n] and Z3=ZA3H.S[n]
151.    // -- Z0, Z2 = {ReRe-ImIm; ReRe-ImIm; ...}
152.    // -- Z1, Z3 = {ImRe+ReIm; ImRe+ReIm; ...}
153.    mova    {z0.b-z3.b}, za0h.b[w14, 0:3]
154.    cmp     w14, w7
155.    b.mi    .Loop_store
156.
157. // Epilogue
158.    fcvtn   z8.h, {z0.s, z1.s}              // Convert and interleave
159.    fcvtn   z9.h, {z2.s, z3.s}              // Convert and interleave
160.
161.    psel    pn10, pn9, p2.b[w14, 0]
162.    st1w    { z8.s-z9.s }, pn10, [x16]      // Store 2 vectors to c_ptr
163.
164.    addvl   x12, x12, #2                    // c_base += 2*SVLb
165.    addvl   x19, x19, #2                    // b_base += 2*SVLb
166.    whilelt pn9.b, x19, x13, vlx2           // N dimension predicate-as-counter
167.    b.first .Loop_N
168.
169.    add     x3, x3, x18           // &matLeft += K*SVLs Cplx-FP16 elems
170.    add     x5, x5, x6            // &matResult_opt += SVLs*ldc Cplx-FP16 elems
171.    incw    x9                    // Loop_M counter increment
172.    whilelt p2.s, x9, x0          // M dimension predicate
173.    b.first .Loop_M
174.
175.    smstop
176.
177.    ldr     x19, [sp], #16
178.
179.    ret
```

9.6 cplx_matmul_opt function overview

The matrix multiplication function uses widening half-precision to single-precision `fmopa` instructions, accumulating to single-precision results in 32-bit ZA tiles, to perform the complex-valued matrix-by-matrix multiplications. Two different 32-bit ZA tiles are used to compute the Real and Imaginary parts of the result:

1. ZA0 and ZA2 accumulate the real part of the result, calculated as `ReA*ReB - ImA*ImB`.
2. ZA1 and ZA3 accumulate the imaginary part of the result, calculated as `ReA*ImB + ImA*ReB`.

Half-precision widening `fmopa` instructions accumulate two-way dot products to each single-precision result, computed as above with one `fmopa` for each part of the result. To accomplish this, the inputs need to be carefully manipulated by preprocessing. This results in a higher computation intensity than was seen in the matmul_fp32 example.

Preprocessing data before the outer product is done using the `revh` and `fneg` instructions:

- `revh`: Reverse every two consecutive 16-bits elements. The resulting vector has the form {Im, Re, Im, Re, ...}.
- `fneg`: Predicated half-precision floating-point negate instruction. Predicates are generated at the start of the function, with only even elements activated. The resulting vector has the form: {Re, -Im, Re, -Im, ...}.

The code uses 1 x 2 tiling, consuming a maximum of 1xSVLs from the N dimension and 2xSVLs from the M dimension in `loop_k`. Two ZA tiles are used for each vector consumed from each dimension.

9.7 cplx_matmul_opt function details

This section describes how the `cplx_matmul_opt` function operates, looking at sections of the code in turn.

- Lines 80, 82, 92, and 95 (Real part computation: `ReA*ReB - ImA*ImB`)

  For the first part of the computation, ZA0.s produces the sum of two outer-products of the two vectors:

  - A negated vector loaded from the transposed `matLeft_mod` matrix in Z8 or Z9.

    The `fneg` instructions at lines 72 and 102 negate only the odd 16-bit lanes of the source vector (the Imaginary parts of the `matLeft_mod` matrix) keeping the even 16-bit lanes unchanged (the Real parts of the `matLeft_mod` matrix). This is enabled by careful prediction using `p6` set at line 47.
  - The first vector loaded from the `matRight` matrix in Z2 or Z6.

  For example, the first row of ZA0.s contains the following value as a result of the first outer-product instruction:

  ```
  Z9.h = {ReA(0,0); -ImA(0,0) ; ReA(1,0) ; -ImA(1,0) ; ...}  // 16-bits Z view
  Z2.h = {ReB(0,0) ; ImB(0,0) ; ReB(1,0) ; ImB(1,0) ; ...}   // 16-bits Z view
  ZA0.s[0] = {ReA(0,0)* ReB(0,0) - ImA(0,0)*ImB(0,0); ...}   // 32-bits view
  ```

  Similarly, the second part of the computation in ZA2 uses the negated data from the `matLeft_mod` transposed matrix with the second loaded vector from the `matRight` matrix in Z3 or Z7.

  Therefore, tiles ZA0.s and ZA2.s accumulate the Real part of the complex-valued results.
- Lines 85, 89, 97, and 100 (Imaginary part computation: `ReA*ImB + ImA*ReB`)

  For the first part of the computation, ZA1.s produces the sum of two outer-products of the two vectors:

  - 16-bit reversed vector loaded from the transposed `matLeft_mod` matrix in Z1 or Z5.

    The `rev` instructions at lines 71 and 98 reverse the content of the 16-bit lanes within each of 32-bit lanes.
  - The first vector loaded from the `matRight` matrix in Z2 or Z6.

For example, the first row of ZA1.s contains the following value as a result of the first outer-product instruction:

```
Z1.h = {ImA(0,0); ReA(0,0) ; ImA(1,0) ; ReA(1,0) ; ...}    // 16-bits Z view
Z2.h = {ReB(0,0) ; ImB(0,0) ; ReB(1,0) ; ImB(1,0) ; ...}   // 16-bits Z view
ZA1.s[0] = {ImA(0,0)* ReB(0,0) + ReA(0,0)*ImB(0,0); ...}   // 32-bits view
```

Similarly, the second part of the computation in ZA3 uses the 16-bit reversed data from the `matLeft_mod` transposed matrix with the second loaded vector from the `matRight` matrix in Z3 or Z7.

Therefore, tiles ZA1.s and ZA3.s accumulate the Imaginary part of the complex-valued results.

- Line 142 (`Loop_store`):

  The Real and Imaginary parts of each result are available in two different 32-bit ZA tiles:

  - First N dimension SVLs into ZA0 (real part) and ZA1 (imaginary part).
  - Second N dimension SVLs elements into ZA2 (real part) and ZA3 (imaginary part).

  `Loop_store` does the following:

  1. Extract the pair of real and imaginary parts from two consecutive 32b ZA tiles.

     The 8-bit `mova` instructions in lines 140 and 153 move four vectors from the four ZA tiles.
  2. Down-convert to half-precision floating point and interleave.

     The `fcvtn` instructions in lines 143 and 144 convert and interleave each of the two consecutive vectors:

     ```
     Z8 and Z9 = {re, Im, re, Im, ...}      // 16-bits Z containers
     ```

     Each `fcvtn` instruction takes a group of two Z vector registers of single-precision elements as input, converts these input elements to half-precision elements, and interleaves elements from both source vectors into the destination vector. This groups the Real and Imaginary parts of each result pair in the 32-bit Z vector registers.
  3. Store the resulting two vectors to the memory.

     The `st1w` instruction at line 146 stores the two consecutive vector results to the result matrix, processing 2 x SVLs result values on the N dimension.

10. Related information

The following resources are related to material in this guide:

- Arm Architecture Reference Manual for A-profile architecture, which contains a specification of the SME Architecture.
- Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)
- Arm C Language Extension
- Fast Model Reference Guide